2008

Metadata And Data Management In High Performance File And Storage Systems

Peng Gu
University of Central Florida

Metadata and Data Management in High Performance File and Storage Systems

by

Peng Gu
B.S. Huazhong University of Science and Technology, 2000
M.S. Huazhong University of Science and Technology, 2003

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2008

Major Professor:
Jun Wang

Abstract

With the advent of emerging "e-Science" applications, today's scientific research increasingly relies on petascale-and-beyond computing over large data sets of the same magnitude. While the computational power of supercomputers has recently entered the era of petascale, the performance of their storage system is far lagged behind by many orders of magnitude. This places an imperative demand on revolutionizing their underlying I/O systems, on which the management of both metadata and data is deemed to have significant performance implications.

Prefetching/caching and data locality awareness optimizations, as conventional and effective management techniques for metadata and data I/O performance enhancement, still play their crucial roles in current parallel and distributed file systems. In this study, we examine the limitations of existing prefetching/caching techniques and explore the untapped potentials of data locality optimization techniques in the new era of petascale computing.

For metadata I/O access, we propose a novel weighted-graph-based prefetching technique, built on both direct and indirect successor relationship, to reap performance benefit from prefetching specifically for clustered metadata serversan arrangement envisioned necessary for petabyte scale distributed storage systems.

For data I/O access, we design and implement Segment-structured On-disk data Grouping and Prefetching (SOGP), a combined prefetching and data placement technique to boost the local data read performance for parallel file systems, especially for those applications with partially overlapped access patterns. One high-performance local I/O software package in SOGP work for Parallel Virtual File System in the number of about 2000 C lines was released to Argonne National Laboratory in 2007 for potential integration into the production mode.

Dedicated to

my wife Yanling Xiao

and

my parents Deying Liu and Ziguo Gu

Acknowledgments

Many thanks go to my advisor Jun Wang for his continued guidance, support, and encouragement. His vision and enthusiasm have served as an inspiration throughout my stay at Nebraska and Florida. He has been a great source of insight and an excellent sounding board for ideas. I am grateful to him for the tremendous time, energy, and wisdom that he invested in steering my research. He has been an excellent role model through his dedication to quality research.

I also thank my thesis committee members Ronald F. DeMara, Shaojie Zhang and David M. Nickerson for reading through this lengthy dissertation. Their detailed comments helped improve the content and presentation of this dissertation.

I am also grateful to Hong Jiang from University of Nebraska, Yifeng Zhu from University of Maine, Robert Ross, Rajeev Thakur, Robert Latham, and Sam Lang from Argonne National Laboratory, for their continued encouragement and support in my efforts to become a good researcher.

I thank the other members of the CASS group, especially Hailong Cai, Dong Li, Xiaoyu Yao, Huijun Zhu, Saba Sehrish, Ping Ge, Pengju Shang, Christopher Mitchell and Grant Mackey for the hours of discussion on topics ranging from movies to file system traces analysis which provided food for thought.

My special thank goes to my family. My wife, Yanling Xiao, has been offering so much help and support to me ever since I met her many years ago. Her parents, also my parent-in-laws, provided great help in may ways. Without their supports, this dissertation would not have been possible.

For all I am, I am indebted to my parents.

This work was supported in part by the US National Science Foundation under Grants CNS-0646910 and CNS-0646911 and CCF-0621526, the US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747, and SEECS@UCF foundation and building fund. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The last five decades have seen an significant increase in processor clock speeds. During this period of time, the performance of individual processors has gone up by 4-5 orders of magnitude. Moreover, high performance computing systems equipped with multi-core or even many-core become increasingly prevailing in both industry and research fields, providing unforseen abundant computational resources to high performance computing system users. On the I/O side, however, although the disk transfer rate and seek time has achieved major improvement in the last five decade as well, the increase is only within two orders of magnitude. This means that it takes two orders of magnitude more disk drives per CPU to do the same relative workload in a balanced way than 50 years ago [RFL06].

Meanwhile, in recent years an increase in the number of Data-Intensive Super Computing (DISC) systems is emerging. These DISC systems differ from conventional supercomputers in their focus on data: they acquire and maintain continually changing data set, in addition to performing large-scale computations over the data. With the massive amount of data arising from such diverse sources as telescope image, medical records, online transaction records, and web pages, DISC systems have the potential to achieve major advances in science, health

care, business efficiencies, and information access [Bry07]. However, in these supercomputing system, I/O performance became largely overlooked or insufficiently emphasized, especially for DISC applications that relies heavily on the data access performance.

Due to the imbalanced development of processing power and I/O capability as well as even higher demands on I/O performance placed by DISC applications, all of this information has some interesting effects on the I/O and file Systems services:

- As the numbers of processing elements goes up, the required file system metadata operations per second goes up, which implies orchestrating metadata requests for more clients than ever due to the slow growth in disk agility.

- Since the number of processing elements is going up rapidly and the amount of memory per processing element is doing down, the I/O system must orchestrate collecting memory from far more memories and issuing more contiguous disk accesses, due to the slow growth in disk bandwidth.

To facilitate realization of these goals, in this dissertation we study the I/O performance from two major aspects: metadata access performance and data access performance to resolve the I/O performance challenge.

The rest part of this dissertation is organized as follows. Chapter 2 briefly introduces the background of metadata and data I/O management in the environment of parallel and distributed storage systems. Chapter 3 presents the design of a weighted-graph based prefetching algorithm for metadata I/O performance improvement. Chapter 4 propose the design

and implementation of a segment-structured on-disk grouping and prefetching technique to improve the data I/O performance in a parallel file system. Chapter 5 concludes the contributions of this dissertation by summarizing the overall study and discussing future research work.

CHAPTER 2

BACKGROUND

2.1 Overview Of Data And Metadata

Since our study heavily relies on the definition of data and metadata, we describe the relative terms here for better understanding of the remaining material presented in this study.

In computer systems, user information is physically stored on components, devices, and recording media that retain digital information used for computing for some intervals of time. All these information stored in computer systems are generally referred to as *data*. Data are typically stored in units of various sizes on storage media. These units may be a bit, a byte, a word, a disk sector, or a disk block, etc. On disk based storage systems, the typical name for such a unit is disk block. All the useful user information, i.e., data are thus stored in disk blocks. However, additional information is required to explain, describe, or locate the original data. This additional information is typically called *metadata*.

The exact meaning of data and metdata can vary in different environments. For example, from a digital photo editor's point of view, the array of bits each storing the corresponding color information for a specific pixel on the picture may be considered data, while other information describing the dimension of the picture, resolution, compression rate, color mapping,

color depth, aperture, exposure parameters of the photo in question may be considered metadata. However, from a file system developer's perspective, all these information are simply user data, while the filename and directory information, file creation time, the ownership information, the location of the data blocks on the disk are considered metadata. Similar situations exist in the field of high performance storage system research also. In our study, unless stated otherwise, we have the following definitions for data and metadata.

Data means user information stored on physical disks, typically in disk blocks, usually also referred as block data, file data, or user data. These words will be used interchangeably in this study.

Metadata means the information used by file systems to explain, describe, or locate the user data. These are typically also referred to as file metadata or file system metadata. For different types of file systems, the format of their metadata may be different. Moreover, the attributes are also very different between local file system metadata and distributed file system metadata.

2.2 Data/Metadata Management

2.2.1 Challenges in Data/Metadata Management

Evidenced by the introduction of a supercomputer with sustained speed of 1.026 petaflop/s debuted in June 2008, the performance of their storage system is far lagged behind by many

orders of magnitude. This places an imperative demand on their I/O system, on which the management of metadata/data is deemed to have significant performance implications.

In the meantime, the data intensive scientific applications such as those in astronomy, biometrics, earthquake science, gravitational-wave physics, and others also result in complex and stringent I/O performance demands that are not satisfied by any existing data/metadata management infrastructure. A large scientific collaboration may generate many queries, each involving access to—or supercomputer-class computations on—gigabytes or terabytes of data. Efficient and reliable execution of these queries may require careful management of terabyte caches, gigabits per second data transfer over wide area networks, co-scheduling of data transfers and supercomputer computation, accurate performance estimations to guide the selection of dataset replicas, and other advanced techniques that collectively maximize use of scarce storage, networking and computing resources.

To cater to these I/O performance demands, parallel and distributed systems have become the mainstream in current storage solutions. A major challenge in today's applications is the physical management of data/metadata in the distributed environment. Although the processing power may be available, getting the data to that computational resource may be time consuming and error-prone. Within the computational site, often a cluster, we also distinguish between shared storage and storage local to a computational node. Identifying the location of desired data sets is a challenge in this type of distributed environment.

2.2.2 Prefetching/Caching Techniques

Prefetching/caching, as conventional and effective techniques for metadata/data I/O performance enhancement, still play their crucial roles in current parallel and distributed file systems.

Cache is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (due to longer access time) or to compute, compared to the cost of reading the cache. A cache normally resides between a slow device and a fast device. It may be RAM memory, a disk storage area, or a combination of both. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or recomputing the original data, so that the average access time is shorter.

Cache has proven to be extremely effective in many areas of computing because access patterns in typical computer applications have locality of reference. The algorithms for discarding and updating cache information can get quite complex. An algorithm can evaluate all cache entries to decide which entries should be flushed based on how often those entries are used.

The use of prefetching can be traced back to as early as mid-1960's. At that time, some cache design researchers noticed the benefit of fetching multiple words from main memory into the cache at one time [AW67]. Such benefit is achieved by taking advantage of spatial locality that are exhibited by many applications. Hardware prefetching was implemented

in the IBM 370/168 and Amdahl 470V [Smith 1978]. Recently, software techniques are prevailing in all layers of storage system hierarchy. Besides the deployment in transferring data from main memory to CPU registers, prefetching can also be applied to move data from disk to main memory in many file system designs to prevent CPU stalls, as evident by Young and Shekita's work [YS93], or Patterson and Gibson's work [PG94].

Moreover, in a distributed computing/storage system, where the network latency can also contribute to the stall of processor cycles, prefetching mechanism are further explored [ACR96, KE91] to move data over the network from the data repository node to the computing node where the actual computation on the data is performed. However, the effectiveness of prefetching mechanism heavily relies on the prefetching algorithms' ability to accurately predict the future data requests and move them as close as possible to the destination processor before those are actually needed by the processor. The difficulties arises as data access pattern in distributed file systems are much harder to predict than in local disk based file systems. The difficulties come not only from the increased storage system complexity, but also the intrinsic complicated and irregular accesses that are performed by large scientific application due to the complicated data representation incurred during the process of application parallelization.

2.2.3 Data and metadata management in Ext3 file system

Many large computing facilities employ parallel and distributed file systems such as Lustre or PVFS to provide sustained high I/O performance to serve their ever increasing I/O needs. We notice that both Lustre and PVFS use Ext3 file system as their back-end local file system. In this section, we present the design of Ext3 local file system to help better understanding these parallel file systems in general.

In Ext3 local file system, data are organized in disk blocks. The basic block size could be 1 KB, 2 KB, or 4 KB. File content are stored in those data blocks, and special blocks called super blocks are used to store the file system metadata.

The space in Ext3 is split up in blocks, and organized into block groups, analogous to cylinder groups in the Unix File System. This is done to reduce external fragmentation and minimize the number of disk seeks when reading a large amount of consecutive data. Each block group contains a superblock, the block group bitmap, inode bitmap, followed by the actual data blocks.

The superblock contains important information that is crucial to the booting of the operating system, thus backup copies are made in every block group of each block in the file system. However, only the first copy of it, which is found at the first block of the file system, is used in the booting.

The group descriptor stores the value of the block bitmap, inode bitmap and the start of the inode table for every block group and these, in turn is stored in a group descriptor table.

2.3 PVFS Background

2.3.1 PVFS overview

PVFS is an open-source, scalable parallel file system targeted at production parallel computation environments. It is designed specifically to scale to very large numbers of clients and servers. The architecture is very modular, allowing for easy inclusion of new hardware support and new algorithms. This makes PVFS a perfect research testbed as well. As a typical representative of parallel file server architectures, PVFS is designed for use in large scale clusters, which scale to petabytes of storage and provide access rates at 100s of GB/s.

2.3.2 PVFS architecture

There are two schools of thought on building parallel file systems: shared storage architectures and parallel file server architectures. The former model is generally considered a high-end solution, featured in higher I/O system bandwidth and higher cost. Whereas, the lack of scalability due to the use of centralized shared storage become its intrinsic deficiency; On the other hand, the latter model (shown in Figure 2.1) is often less reliable, but more scalable and cost-effective, and thereby suitable for low-end solutions. As a result, PVFS adopts the parallel file server architecture. More specifically, PVFS use a client/server architecture to manage its data and metadata. Both the server and client side libraries can reside completely in user space. Clients initiate requests for file accesses with one of the servers.

The actual file IO is striped across a number of file servers. Storage spaces of PVFS are managed by and exported from individual servers using native file systems available on the local nodes.

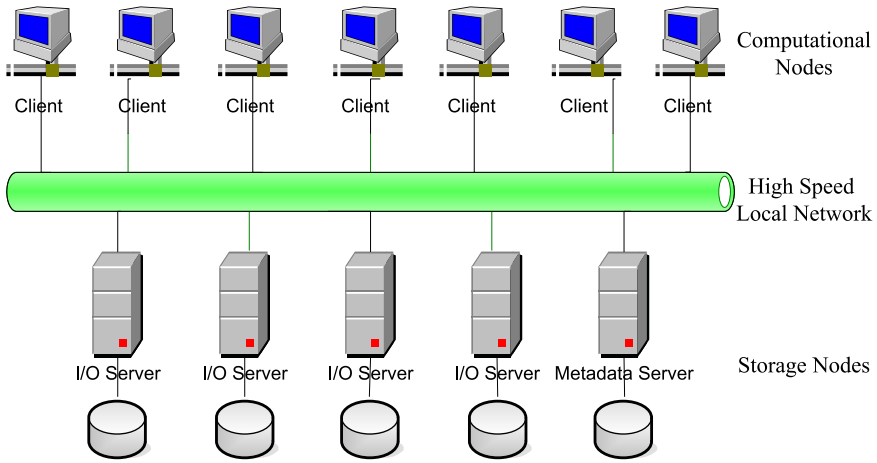


Figure 2.1: PVFS Architecture

2.3.3 PVFS I/O path

There are two ways for a client to retrieve data from a PVFS I/O server, using either a VFS system interface or a library call with libpvfs and MPI-IO [CFH94] library. In the first approach, the PVFS file system is mounted on the client side. This allows users to change and list directories, move files, and execute binaries from the file system as well, just as they normally do in a traditional UNIX file system. This mechanism introduces some performance

overhead but seems to be the most convenient way to access the file interactively. In order to take the advantage of PVFS, many scientific applications prefer the second interface — MPI-IO. The MPI-IO interface helps optimize access to individual files by enabling many processes running simultaneously on different nodes. It also provides a "noncontiguous" access operation that allows for efficient access to data spread throughout the whole file.

CHAPTER 3

METADATA PREFETCHING IN LARGE SCALE DISTRIBUTED STORAGE SYSTEM

3.1 Chapter Overview

Although data prefetching algorithms have been extensively studied for years, there is no counterpart research done for metadata access performance. Existing data prefetching algorithms, either lack of emphasis on group prefetching, or bearing a high level of computational complexity, do not work well with metadata prefetching cases. Therefore, an efficient, accurate and distributed metadata-oriented prefetching scheme is critical to leverage the overall performance in large distributed storage systems. In this chapter, we present a novel weighted-graph-based prefetching technique, built on both direct and indirect successor relationship, to reap performance benefit from prefetching specifically for clustered metadata servers, an arrangement envisioned necessary for petabyte scale distributed storage systems. Extensive trace-driven simulations show that by adopting our new metadata prefetching algorithm, the miss rate for metadata accesses on the client site can be effectively reduced, while the average response time of metadata operations can be dramatically cut by up to

67%, compared with legacy LRU caching algorithm and existing state of the art prefetching algorithms.

The outline of the rest of the chapter is as follows: Related work is discussed in Section 3.3. Section 3.4 shows the fundamental difference between data and metadata size distribution. Section 3.5 describes our Nexus algorithm in detail. Evaluation methodologies and results are discussed in section 4.5. We conclude this chapter in section 4.6.

3.2 Motivation

A novel decoupled storage architecture diverting actual file data flows away from metadata traffic has emerged to be an effective approach to alleviate the I/O bottleneck in modern storage systems [Sch03, ZJW04, GGL03, WBM06a]. Unlike conventional storage systems, these new storage architectures use separate servers for *data* and *metadata* services, respectively, as shown in Figure 3.1.

Accordingly, large volume of actual file data does not need to be transferred through metadata servers, which significantly increases the data throughput. Previous studies on this new storage architecture mainly focus on optimizing the scalability and efficiency of file *data* accesses by using a RAID style striping [Had00, HO95], caching [ORR04], scheduling [GA03] and networking [MCM01]. Only recent years have seen growing activities in studying the scalability of the metadata management [ZJW04, GZJ06, WBM06b, DW07]. However, the performance of metadata services plays a critical role in achieving high I/O scalability

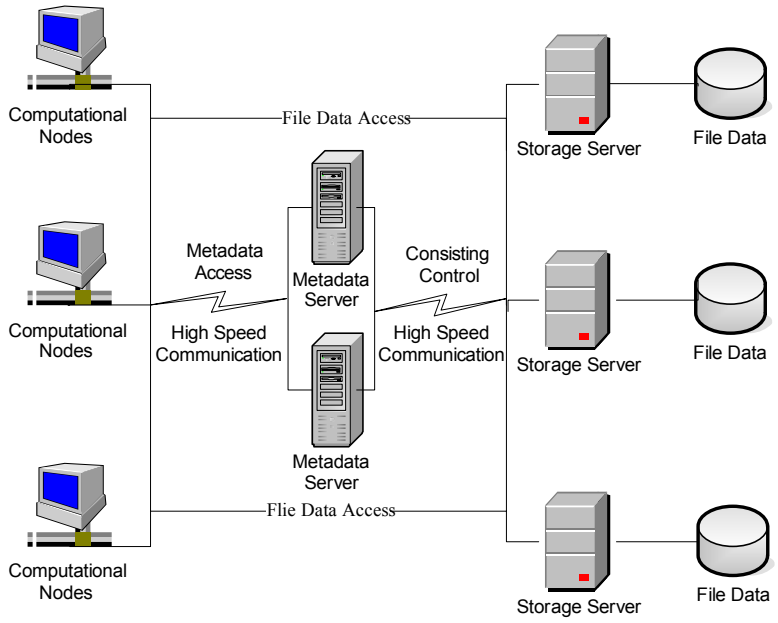


Figure 3.1: System architecture

and throughput, especially in light of the rapidly increasing scale in modern storage systems for various data intensive supercomputing applications, such as predicting and modeling the effects of earthquakes and web search without language barriers. In these applications the volume of data reaches and even exceeds Peta bytes (10^{15} bytes) while metadata amounts to Tera bytes (10^{12} bytes) or more [WPB04]. In fact, more than 50% of all I/O operations are to metadata [RLA00], suggesting further that multiple metadata servers are required for a petabyte-scale storage system to avoid potential performance bottleneck on a centralized metadata server. This study takes advantages of some unique characteristics of metadata and proposes a new prefetching scheme particularly for metadata accesses that is able to scale up the performance of metadata services in large scale storage systems.

By exploiting the access locality widely exhibited in most I/O workloads, caching and prefetching have become an effective approach to boost I/O performance by absorbing a large number of I/O operations before they touch disk surfaces. However, existing caching and prefetching algorithms may not work well for metadata since most caching and prefetching schemes are designed for and tested on actual file data and simply ignore metadata characteristics. As a result of this negligence, traditional caching and prefetching algorithms are not specifically optimized for metadata. And thus they may consequently not fit well with metadata access cases because file data and metadata operations usually have different characteristics and exhibit different access behaviors. For example, a file might be read multiple times while its metadata is only accessed once. An "ls -l" command touches the metadata of multiple files but might not access their data. In addition, the size of metadata is typically uniform and much smaller than the size of file data in most file systems (regarding this point, we will show further elucidation in section 3.4). With a relatively small data size, the misprefetching penalty for metadata on both the disk side and the memory cache side is likely much less than that for file data, allowing the opportunity for exploring and adopting more aggressive prefetching algorithms. In contrast, most of the previous prefetching algorithms share the same characteristic in that they are conservative on prefetching. They typically prefetch at most one file upon each cache miss. Moreover, even when a cache miss happens, certain rigid policies are enforced before issuing a prefetching in order to maintain a high level of prefetching accuracy. The bottom line is, they did not realized that considering the

huge number and the relatively small size of metadata items, aggressive prefetching can be profitable.

On the other hand, aggressive prefetching or group-based prefetching can easily balance out their advantages by introducing 1) extra burden to the disk, 2) cache pollution and 3) high CPU runtime overhead. Hence, part of the challenges in developing an aggressive prefetching algorithm is to address the three problems at the same time.

3.3 Related Work

Prefetching and caching has long been studied and implemented in modern file systems. In the area of disk level and file level prefetching, most of previous work were done in three major areas: predictive prefetching [KE93, LD97], application controlled prefetching [Tom97, PGS93, CFK96, KL91], and compiler directed I/O [SD00, MDK96]. The latter two have limited applicability due to their constraints. For example, application controlled prefetching requires source code revision and compiler directed I/O relies on sufficient time intervals between prefetching instructions inserted by compiler and the following actual I/O instructions. Since predictive prefetching, using past access pattern to predict future access, is completely transparent to clients, it is more suitable for general practice, including metadata prefetching.

Unfortunately, although the split data-metadata storage system became ever popular for providing large scale storage solutions, there is a general negligence on the study of prefetch-

ing algorithms specifically for metadata servers: current predictive prefetching algorithms are for data but not metadata.

In order to better illustrate the difference between our Nexus algorithm and other predictive prefetching algorithms, next we briefly introduce some background in this field.

On prefetching objects in object-oriented database, Curewitz developed a probabilistic approach [CKV93]. On prefetching whole files, Griffioen and Appleton introduced a probability graph based approach to study file access patterns [GA94]. In addition, Duchamp *et al.* studied an access tree based prediction approach [LD97]. However, all the above approaches only consider immediate successors relationship in their study other than indirect successors relationship. The advantages of approaches considering both immediate and subsequent successor relationships are discussed in detail in section 3.5.

Based on the previous research, Long *et al.* developed a serial of successor-based predictive prefetching algorithms in their efforts to advance the prefetching accuracy while maintaining a reasonable performance gain [KL99, AL01, AB02, KL01]. The features of these predictors are summarized below as they are state of the art and are most relevant to our design.

First Successor [AL01] The file that followed file A the first time A was accessed is always predicted to follow A.

Last Successor [AL01] The file that followed file A the last time A was accessed is predicted to follow A.

Noah (Stable Successor) [AL01] Similar to Last Successor, except that a current prediction is maintained; and the current prediction is changed to last successor if last successor was the same for S consecutive accesses where S is a predefined parameter.

Recent Popularity (Best j-out-of-k) [AB02] Based on last k observations on file A's successors, if j out of those k observations turn out to target the same file B, then B will be predicted to follow A.

Probability-based Successor Group Prediction [AB02] Based on file successor observations, a file relationship graph is built to represent the probability of a given file following another. Based on the relationship graph, the prefetch strategy builds the prefetching group with a predefined size S by following steps:

1. The missed item is first added into the group.
2. Add the items with the highest conditional probability under the condition the items in the current prefetching group were accessed together.
3. Repeat step 2 until the group size limitation S is met.

Among the aforementioned five predictors, the three former ones fall into the category of single-successor predictor. The two latter predictors, although being group-based predictors,

if revised to take additional indirect successors into consideration for relationship graph construction, would inevitably introduce exponential time overhead. The detail is explained in 3.5.4.2.

3.4 File Data And Metadata Size Distribution

We first explain how we obtain the file data distribution information on Franklin's Lustre file system in Section 3.4.1. Based on these information, we then explain how we estimate the corresponding Ext3 inode metadata size distribution in Section 3.4.2. The method we used to collect the directory size distribution information is described in Section 3.4.3. Based on all these data and metadata size distribution information, a comparison between data size distribution and metadata size distribution is then presented in Section 3.4.4.

3.4.1 Obtaining file data size distribution

To find out the difference between file data and metadata size distribution, we studied the files stored on Franklin supercomputer. Franklin is a massively parallel processing (MPP) system with 9,660 compute nodes, serving more than 1,000 users at National Energy Research Scientific Computing Center. The collection of file size distribution is somewhat straight forward compared with the metadata size case. We simply run a "ls -lR /" on the head node and then use a script to filter out the file size information from the output. Note that since we do not have the privilege to access all the files and directories stored on the system,

by running these scripts we only get the size information of those files and directories that are accessible. In this study, we collected the size information for 8,209,710 regular files and 612,248 directories. The cumulative distribution function (CDF) of collected file size distribution results is shown in Figure 3.2.

3.4.2 Obtaining metadata size distribution

Obtaining the metadata size information is not as simple. To the best of our knowledge, there is no direct way/utility in existence to find out the metadata size information for files and directories. However, there does exist a way of figuring out the corresponding metadata size if we know the file size (assuming file system type and the block size are given). For example, in an Ext2 file system, the metadata of a regular file consist of two components: a mandatory inode block and conditional indirect addressing blocks. According the to latest Linux kernel source code as of this writing (version 2.6.25.9 released on June 24, 2008 [Lin07]), each Ext2 inode structure is 128 bytes in length. This inode structure contains 12 direct block pointers plus 1 indirect block pointer, 1 double indirect block pointer and 1 triple indirect block pointer [BC05]. Once the block size is given, we are able to calculate the on-disk space occupied by indirect addressing blocks for files of any given size. The resulting metadata size is then the sum of the inode block size and the space for indirect addressing blocks. The detailed size mapping information between file data and metadata is summarized in Table 3.1.

Table 3.1: Size conversion between file data and metadata

Block size	**Addressing Mode**	**File size**	**Metadata size**
1024	Direct	$\leq$ 12 KB	128
	1-Indirect	12 KB$\sim$268 KB	1152
	2-Indirect	268 KB$\sim$64.26 MB	3200
	3-Indirect	64.26 MB$\sim$16.06 GB	6272
2048	Direct	$\leq$24 KB	128
	1-Indirect	24 KB$\sim$1.02 MB	2176
	2-Indirect	1.02 MB$\sim$513.02 MB	6272
	3-Indirect	513.02 MB$\sim$256.5 GB	12416
4096	Direct	$\leq$48 KB	128
	1-Indirect	48 KB$\sim$4.04 MB	**4224**
	2-Indirect	4.04 MB$\sim$4 GB	12416
	3-Indirect	4 GB$\sim$4 TB	24704

Note that Franklin's user home directory uses Lustre file system, which subsequently uses Ext3 file sytem as its back-end file system [Mic08]. Furthermore, Ext3 file system's data structures on disk are essentially identical to those of an Ext2 file system, except that it employs a journal to log metadata and data changes. This means the method we just described can be applied to calculate the metadata size based on the file data size

collected. For example, given a block size of 4 KBytes (which is the basic block size for back-end Ext3 file systems chosen by Lustre file system developers) and a file size of 3 MB, the corresponding metadata size will be 4224 bytes (highlighted in Table 3.1). Note that although this calculation applies only to Ext2 or Ext3 local file system, similar calculations can be applied to and similar conclusions can be drawn for other parallel or distributed file systems such as GPFS [SH02], PVFS, Panasas [WUA08], and Ceph. All of these file systems use some forms of pointers to refer to certain chunk(s) of data, no matter these data are stored as regular blocks, local files, or objects.

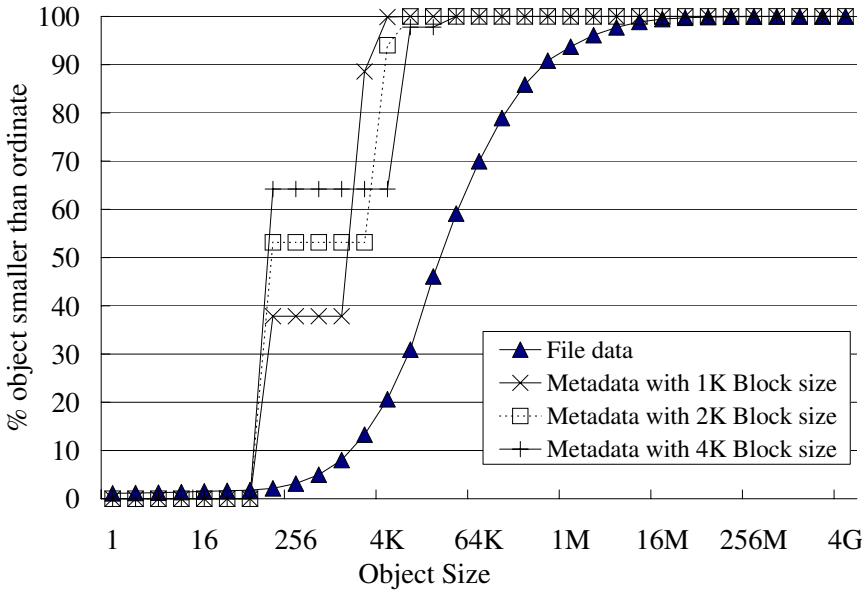


Figure 3.2: Size distribution comparison of file data and metadata

According to the file size distribution, we obtain the corresponding metadata size distribution for the files, and the results are shown together with the file size distribution in Figure 3.2 for ease of comparison.

3.4.3 Directory size distribution

Metadata include both file inodes and directories: we have so far discussed the metadata size for file inodes, It may also be interesting to find out the directory size distribution. Directories are organized the same way as regular files in Linux-based system. By *directory size* we simply mean the space in bytes occupied by those file names under the directory. To obtain directory size for certain directory, we iterrate all the files and sub-directories under that directory and sum up the length of all the file names and sub-directory names[1]. The corresponding results are shown in Figure 3.3. According to these results, around 95% of the directory sizes are less than 600 bytes.

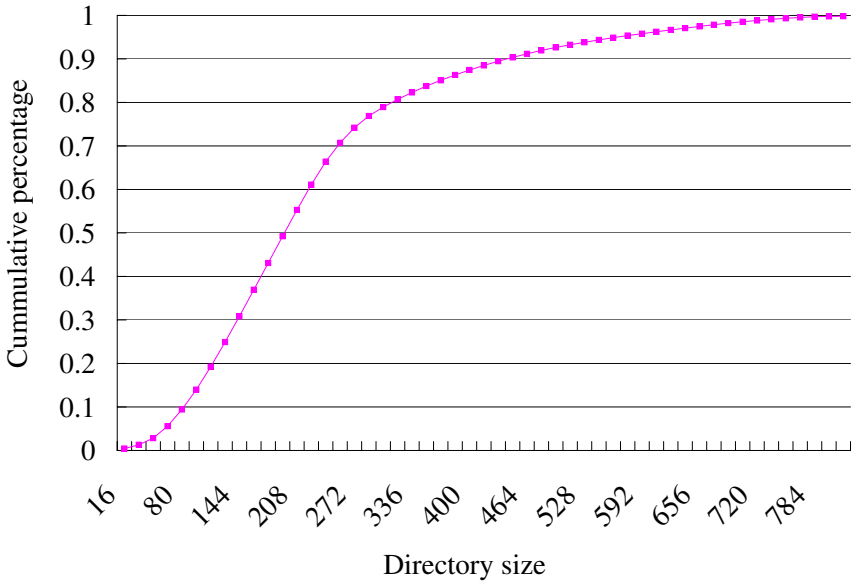
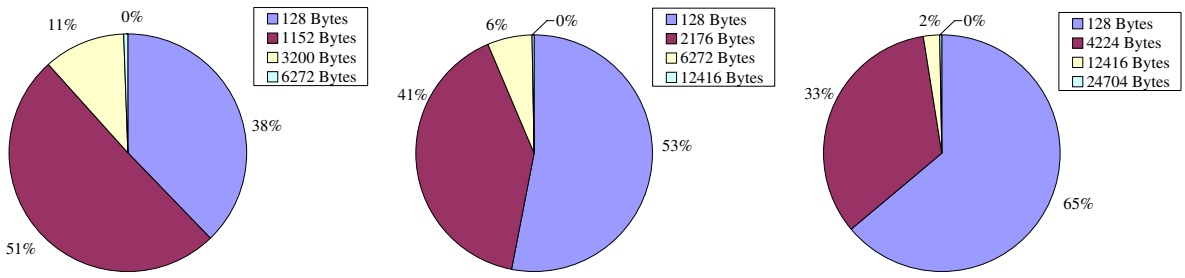


Figure 3.3: Directory size distribution

[1]the length of file name including a ending '\0' should be rounded/aligned to a multiple of four bytes, which is an optimization done in Ext2 file system implementation.



(a) Block size = 1024 Bytes (b) Block size = 2048 Bytes (c) Block size = 4096 Bytes

Figure 3.4: Metadata size distribution

3.4.4 Comparison

The study of distinction between data and metadata size distribution for all the files on Franklin supercomputer reveals the results shown in Figure 3.2. We observe that the file size distribution and metadata size distribution are quite different. For both data and metadata that are less than 64 bytes, the percentage is very small, i.e. less than 2%. However, around 71% of files are larger than 8 KBytes; while more than 97% of metadata are smaller than 8 KBytes under all three different block sizes. Moreover, Figure 3.4 shows the exact size distribution of the metadata under different block sizes in a more direct and conspicuous way. Specifically, Figure 3.4(a) shows that for 1 KBytes block size, 89% of metadata are less than or equal to 1152 bytes. Figure 3.4(b) shows that for 2 KB block size, 94% of metadata are 2176 bytes or less. Figure 3.4(c) shows that for 4 KB block size, the percentage of metadata sizes larger than 4224 bytes is almost negligible.

Based on our file data and metadata size distribution research, we observe that compared with typical file size, metadata are relatively small. These results are collected from NERSCs Franklin supercomputer equipped with over 350 TBytes of usable storage. We envision that the same conclusion holds for petabyte scale storage system if there is no significant change on the way the file systems manage their data and metadata. Consequently, in order to achieve optimal performance, a new prefetching algorithm that considers the size differences between data and metadata is clearly desirable. And a good example to be considered is an aggressive prefetching scheme.

3.5 NEXUS: A Weighted-Graph-Based Prefetching Algorithm

As a more effective way for metadata prefetching, our Nexus algorithm distinguishes itself in three aspects. First, Nexus can more accurately capture the metadata access temporal locality exhibited in metadata access streams by observing the affinity among both immediate and subsequent successors. Second, Nexus exploits the fact that metadata usually is small in size and deploy an aggressive prefetching strategy. Third, Nexus maintains a polynomial runtime overhead.

3.5.1 Relationship graph overview

Our algorithm uses a metadata relationship graph to assist prefetching decision making. The relationship graph is used to dynamically represent the locality strength between predeces-

sors and successors in metadata access streams. Directed graphs are chosen to represent the relationship since the relationship between a predecessor and a successor is essentially unidirectional. Each metadata corresponding to a file or directory is represented as a vertex in our relationship graph. The locality strength between a pair of metadata items is represented as a weighed edge. To illustrate this design, Figure 3.5 shows an artificially simplified example of relationship graph consisting of metadata for six files/directories. An observation obtained on this toy example is that the predecessor-successor relationship between $/usr$ and $/usr/bin$ is much stronger than that between $/usr$ and $/usr/src$.

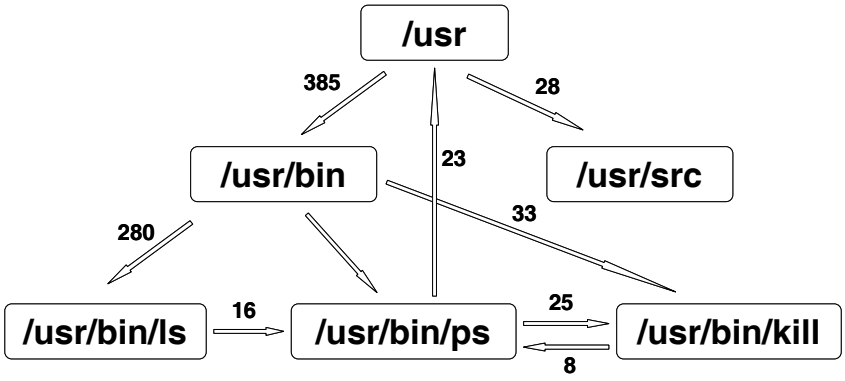


Figure 3.5: Relationship graph demo

3.5.2 Relationship graph construction

To understand how this relationship graph works for improved prefetching performance, it is necessary to first understand how this graph is built. The relationship graph is built on the fly while the MDS receives and serves requests from a large number of clients. A look-ahead

history window with a predefined capacity is used to keep the requests most recently received by the MDS server.

For example, if the history window capacity is set to ten, only ten most recent requests are kept in the history window. Upon the arrival of a new request, the oldest request in this history window is replaced by the new comer. In this way the history window is dynamically updated and always contains the current predecessor-successor relationship at any time. The relationship information is then integrated into the graph on a per-request basis, by either inserting a new edge (if the predecessor-successor relationship is discovered for the very first time) or add appropriate weight to an existing edge (if this relationship has been observed before). A piece of pseudocode describing how the relationship graph is built from the beginning is provided in Table 3.2 and an example is given in Figure 3.6 for better understanding.

In this example, an sample request sequence of

$$ABCADCBA\cdots$$

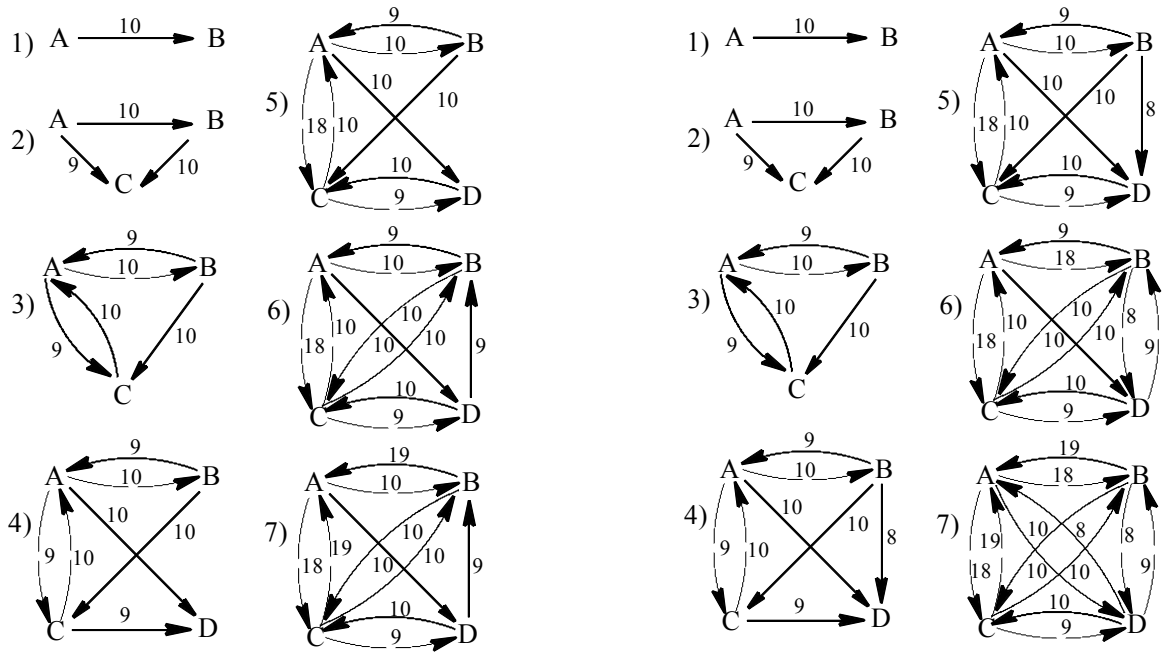
is given. Figure 3.6(a) shows the step by step graph construction from scratch with a history window size of two (The weight assignment methodology assumed here is linear decremental, described later in Section 3.5.5.2 on page 37). In contrast, Figure 3.6(b) shows the same relationship graph construction procedure with a history window size of three.

Table 3.2: Nexus grouping algorithm pseudocode

```
// Let G denote the graph to be built
BUILD-RELATIONSHIP-GRAPH(G)
1   G ← ∅
2   for each new incoming metadata request j
3     for each metadata request i (i ≠ j) in history window
4       if edge (i, j) ∉ G
5       then add an edge (i, j) to G with appropriate weight
6       else adding appropriate weight to edge (i, j)
7     replace the oldest item in history window with j
```

3.5.3 Prefetching based on the relationship graph

Once the graph is built for the access sequence $ABCADCBA\cdots$ as shown in Figure 3.6(a) or Figure 3.6(b), we are now ready to prefetch a successor group with an adjustable size in the graph when a cache miss happens for an element in that group. The prediction result depends on the order of the weights (represented by numbers associated with arrows in Figure 3.6) of outbound edges originated from the latest missed element. A larger weight indicates a closer relationship and a higher prefetching priority. Assuming the last request A in the above sample access sequence sees a miss, according to the graph shown in Figure 3.6(a), the prediction result will be $\{C\}$ if the prefetching group size is one, or $\{C,D\}$



(a) Look-ahead history window size = 2

(b) Look-ahead history window size = 3

Figure 3.6: Graph construction examples

if the prefetching group size is two; similar results deduced from Figure 3.6(b) will be $\{B\}$ and $\{B, C\}$, respectively (as shown in Table 3.3).

3.5.4 Major advantages of Nexus

3.5.4.1 The farther the sight, the wiser the decision

The key difference between the relationship-based and probability-based approaches lies in the ability to look farther than the immediate successor. The shortcoming of the probability-based prefetching model is obvious: it only considers the immediate successors as candidates

Table 3.3: Prediction Results Comparison. P1 means prefetching with group size = 1; P2 means prefetching with group size = 2; N2 means Nexus with history window size = 2; N3 means Nexus with history window size = 3

	N2	N3
P1	C	B
P2	CD	CB

for future prediction. As a consequence, any successors after the immediate successor are ignored. This short-sighted method is incapable of identifying the affinity of two references with some intervals, which widely exists in many applications. For example, for the pattern "$A?B$", we can easily find two situations where this pattern exhibits.

- Compiling programs: gcc compiler("A") is always first launched; and then the source code("?") to be compiled is loaded; at last the common header files or common shared libraries ("B") is loaded afterward.

- Multimedia application: initially media player application ("A") is launched; after that the media clip ("?") to be played is loaded; at last the decoder program ("B") for that type of media is loaded.

In addition to above mentioned applications, interleaved application I/Os coming from multi-core computers or from many clients will only make things worse. The probability-based model can not detect such access patterns, thus limiting its ability to make better pre-

dictions. However, this omitted information is taken into consideration in our relationship-based prefetching algorithm, which is able to look farther than the immediate successor when we build our relationship graph.

We use the same aforementioned sample trace sequence, $ABCADCBA\cdots$, to further illustrate the difference between the probability-based approach and our relationship-based method. In the probability-based model, since C never appears immediately after A, C will never be predicted as A's successor. In fact, the reference stream shows that C is a good candidate as A's *indirect* successor because it always shows up *next next* to A. The rationale is that the pattern we observed is a repetition of pattern "$A?C$" and thus we predict this pattern will repeat in the near future. As discussed in 3.5.3, should our relationship-based prediction be applied, three out of four prediction results will contain C.

From the above example, we clearly see the advantages of relationship-based prefetching over probability-based prefetching. The essential ability to look farther than the immediate successor directly renders this advantage.

3.5.4.2 Farther sight within small overhead

The aforementioned advantage comes at the cost of a look-ahead history window. This approach appears to be prohibitive for other online prefetching algorithms due to potential high runtime overhead. However, this overhead is kept minimum in our design. In fact,

we actually achieved a polynomial time complexity for our relationship graph construction algorithm as shown in Table 3.2.

Theorem 1. *The Nexus grouping algorithm given in Table 3.2 bears polynomial time complexity.*

Proof. Let L denote the look-ahead history window size; let n denote the length of the entire metadata access history. We will first calculate the time required by each step described in Table 3.2 and then derive the aggregated algorithm complexity. Step 1 always takes constant time, i.e., $O(1)$. Step 2 dictates that step 3 through 7 should be executed n times. Consequently step 3 dictates that step 4 through 6 should run L times. Step 4 requires constant time assuming a two-dimensional adjacency matrix representation is adopted for graph G. Either step 5 or step 6 is chosen to be executed next according to runtime conditions. Since both step 5 and step 6 require constant time, regardless of which one is selected, the result would be the same, i.e., $O(1)$ for step 5 and 6 combined. Step 7 also takes constant time, as it replaces the oldest item by overwriting the array element in the circular history window pointed by the last-element-pointer and shifting that pointer to the next element, thus no scanning or searching is involved. Putting it all together, the time complexity for this algorithm is $O(1) + O(n) \cdot \{O(L) \cdot [O(1) + O(1)] + O(1)\} = O(n \cdot L)$, which means a polynomial time complexity. □

In contrast, should we apply the same idea to a probability-based approach, the complexity of the algorithm would be exponential. For example, if look-ahead history window size

is set to 2 (i.e., L=2) rather than 1 (L=1 means only looking at the immediate successor), a probability-based approach would maintain the conditional probability per 3-tuple $P(C|AB)$ instead of per 2-tuple $P(B|A)$. Under the same assumption for graph representation as used in the proof above, we can prove that the time complexity will be $O(n^L)$ for probability-based approach as opposed to $O(n \cdot L)$ for Nexus. If we choose to switch to adjacency list graph representation for the sake of potential less memory usage[2], the algorithm time complexity would grow to prohibitive $O(n^{L+2})$ for a probability-based approach while only $O(n^3 \cdot L)$ for Nexus. To make it clearer, simply consider an example of $L = 5$ and $n = 1000$, the time complexity difference between two algorithms would be 10^{15} against 5000 with adjacency matrix representation, or 10^{21} against 5×10^9 with adjacency list alternative.

3.5.4.3 Aggressive prefetching is natural for metadata servers

All previous prefetching algorithms tend to be conservative due to the prohibitive mis-prefetch penalty and cache pollution [ZL07]. However, the penalty of an incorrect metadata prefetch might be much less prohibitive than that of the file data prefetch, and the cache pollution problem is not as severe as in the case of file data caching. The evidence is the observation that 99% of metadata are less than 4224 bytes, while 40% of file data are larger than 4 KB, as observed in 3.4.4. On the other hand, we also observe that metadata servers and compute nodes equipped with multiple gigabytes or even terabytes of memory become norm.

[2]Switching to adjacency list representation may reduce memory space occupation at the cost of potential computing time increase if the original adjacency matrix turns out to be a sparse matrix.

These observations encourages us to conduct aggressive prefetching on metadata, considering that a single cache miss at the client site will result in mandatary network round-trip latency plus potential disk operation overhead when the requested metadata server consequently see a cache miss.

3.5.5 Algorithm design considerations

When implementing our algorithms, several design factors need to be considered to optimize the performance. Corresponding sensitivity studies on those factors are carried out as follows.

3.5.5.1 How far to look ahead and how many to prefetch

To fully exploit the benefit of bulk prefetching, we need to decide the distance to look ahead and the bulk size to prefetch. Looking ahead too far may compromise the algorithm's effectiveness by introducing noises to the relationship graph; and prefetching too much may result in a lot of inaccurate prefetching, possible cache pollution, and cause performance degradation. We compare the average response time by performing a number of experiments on a combination of these two key parameters, i.e., look-ahead history window size and prefetching group size. In these experiments we adopt the same simulation framework described in Section 3.6.2. The result is shown in Figure 4.6. From Figure 4.6, we found that looking ahead 5 successive files' metadata and prefetching 2 files' metadata at a time turned out to be the best combination. The results also seem to suggest that the larger the look-ahead

history window size, the better the hit rate achieved. This observation prompts us to experiment on much larger look-ahead history window, with sizes 10, 50, and 100 respectively, and found contradicting results to our conjecture: none of those three look-ahead history window size configurations achieves a better hit rate than the windows size of 5. The reason is that looking too far ahead might overwhelm the prefetching algorithm by introducing too much noise–those irrelevant future accesses are also taken into consideration as successors, reducing the effectiveness of the relationships captured by the look-ahead history window. In the rest of the experiments, the look-ahead distance and the prefetching group size are fixed to 5 and 2 respectively for best performance gains. In addition, since a cache size as small as 10% is good enough to demonstrate this performance gain, we will use this as the default configuration unless otherwise specified.

3.5.5.2 Successor relationship strength

Assigning an appropriate weight between the nodes to represent the strength of their relationship as predecessor and successor is critical to our algorithm because it affects the prediction accuracy of our algorithm. A formulated description of this problem is: Given an access sequence of length n:

$$M_1 M_2 M_3 \dots M_n,$$

how much weight should be added to the predecessor-successors edges,

$$(M_1, M_2), (M_1, M_3), \dots, (M_1, M_n),$$

respectively. Four approaches are taken into consideration:

- *Identical assignment* Assigning all the successors of M_1 the same importance. This approach is very similar to the probability model introduced by Griffioen and Appleton [GA94]. It may look simple and straightforward, but it is indeed effective. The key point is that at least the successor following the immediate successors are taken into consideration. However, the draw back of this approach is also obvious: it cannot differentiate the importance of the immediate successor and its followers, which might subsequently skew the relationship strengths to some extend. This approach is referred to as *identical* assignment for later discussions.

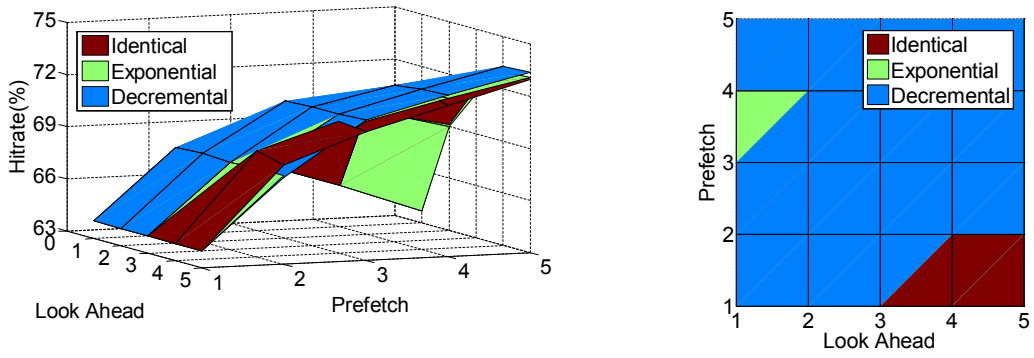
- *Linear decremental assignment* The assumption behind this approach is that the closer the access distance in the reference stream, the stronger the relationship. For example, we may assign those edge weights mentioned above in a linear decremental order, as 10 for (M_1, M_2), 9 for (M_1, M_3), 8 for (M_1, M_4), and so on. (The weight in the example shown in Figure 3.6(a) and Figure 3.6(b) is calculated this way.) This approach is referred to as *decremental* assignment in the rest of this chapter.

- *Polynomial decremental assignment* Another possibility is that, with increase in the successor distance, the decrease in the relationship strength might be more radical than the linear one. For example, polynomial decrement assignment is a possible alternative solution. This assumption is based on the observation of the attenuation of radiation in the air in our real life.

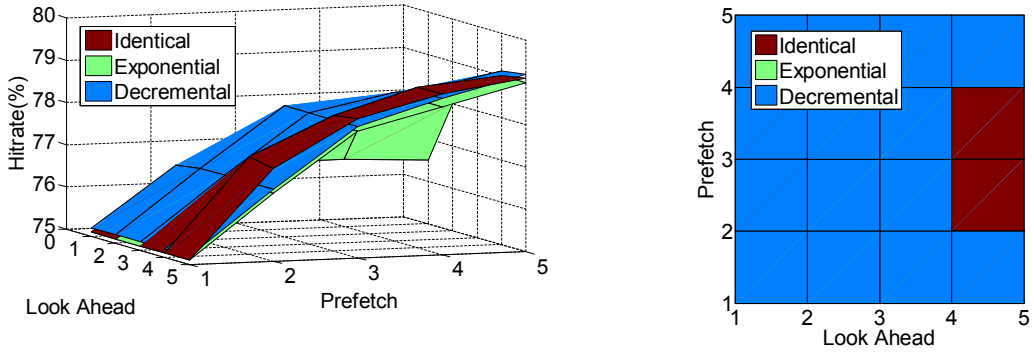
- *Exponential decremental assignment* The attenuation of edge weights might be even faster than polynomial decrement. In this case, an exponential decrement model is adopted. This approach is referred to as *exponential* decremental assignment in the future.

To find out which assignment method can best reflect the locality strength in the metadata reference streams, we conduct experiments on the HP file server trace [RLA00] to compare the hit rate achieved by those four edge-weight assignment methods. To be comprehensive, these experiments are conducted with different configurations in three dimensions: cache size, number of successors to look ahead (or history window size), and number of successors to prefetch as a group (or prefetching group size). In our experiments, the cache size (as a fraction of total metadata workset size) varies from 10% to 90% in an ascending step of 20%. We found that the effects of prefetching become negligible once the cache size exceeds 50%. Accordingly, in this paper, we only presented the results with cache size of 10%, 30% and 50%. In addition, we also observe that the results for the polynomial assignment is very close to those for the exponential assignment, so we remove the former results to show readers a clearer figure. The results for the remaining three approaches are shown in Figure 3.7.

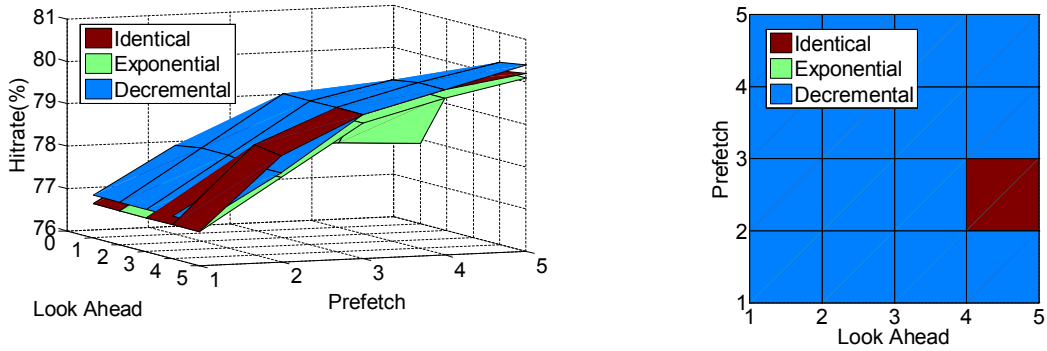
In Figure 3.7, the 3D graphs on the left show the hit rate achieved by those three approaches over three different cache size configurations (i.e. 10%, 30% and 50%) with both the look-ahead history window size and prefetching group size varying from 1 to 5. (The values are carefully chosen in order to be representative while non-exhaustive.) The three



(a) Cache size = 10%

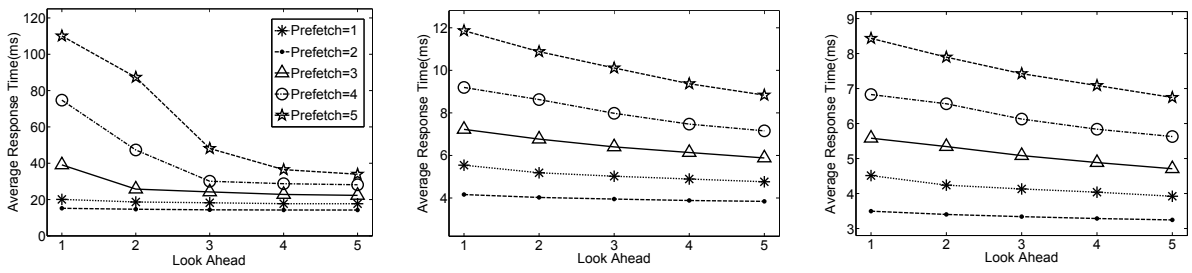


(b) Cache size = 30%



(c) Cache size = 50%

Figure 3.7: Edge weight assignment approaches comparison



(a) Cache Size = 10% (b) Cache Size = 30% (c) Cache Size = 50%

Figure 3.8: Sensitivity Study: Look Ahead and Prefetch Group Size

2D graphs on the right show the corresponding *planform* (a X-Y plane looking downward along the Z axis) of the same measurements. These 2D graphs clearly show that the linear *decremental* assignment approach takes the lead most of the time. We also notice that the identical assignment beats others in some cases even though this approach is very simple. Since the linear decremental assignment approach consistently outperforms others, in the future experiments, we will deploy this approach as our edge-weight-assignment scheme.

3.5.5.3 Server-oriented grouping *vs.* client-oriented grouping

One way to improve the effectiveness of the metadata relationship graph is to enforce better locality. Since multiple client nodes may access any given metadata server simultaneously, most likely request streams from different clients will be interleaved, making the pattern more difficult to observe. Thus it may be a good idea to differentiate the different clients when building the relationship graph. Thus there are two different approaches to build the relationship graph on the metadata servers: 1) Build a single relationship graph for all

the requests received by a particular metadata server; or 2) Build a relationship graph for requests originated from each individual client and received by a particular metadata server. In this study, we refer to the former version as server-oriented access grouping, and the latter as client-oriented access grouping.

We have developed a client-oriented grouping algorithm and compared it with the server-oriented grouping by running them on the HP traces, as shown in Figure 3.9.

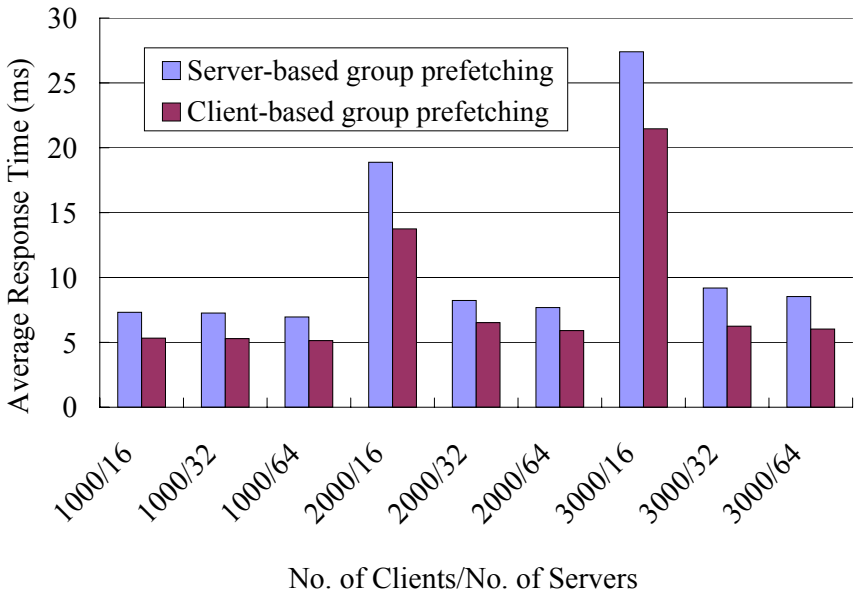


Figure 3.9: Server-Oriented Grouping VS Client-Oriented Grouping

Figure 3.9 clearly shows that client-oriented grouping algorithm consistently outperforms the server-oriented one. Thus we adopt the client-oriented grouping algorithm whenever possible.

3.6 Evaluation Methodology And Results

This section describes the workload, the simulation framework, and the detailed simulation we used to evaluate the metadata performance equipped with Nexus. The metrics we used here include hit rate and average response time. In addition, we also studied the impact of consistency control and scalability of Nexus algorithm.

3.6.1 Workloads

We evaluate our design by running trace-driven simulations over one scientific computing trace and one file server trace: the LLNL trace collected at Lawrence Livermore National Laboratory in July 2003 [WXH04] and the HP-UX server file system trace collected at the University of California Berkeley in December 2000 [RKS02]. These traces gather I/O events of both file data and metadata. In our simulations, we filter out the file data activities and feed only metadata events to our simulator.

3.6.1.1 LLNL trace

One of the main reasons for petabyte-scale storage systems is the need to accommodate scientific applications that are increasingly demanding on I/O and storage capacities and capabilities. As a result, some of the best traces to evaluate our prefetching algorithm are those generated by scientific applications. To the best of our knowledge, the only recent

scientific application trace publicly available for large clusters is the LLNL 2003 file system trace. It was obtained in the Lustre Lite [Sch03] parallel file system on a large Linux cluster with more than 800 dual-processor nodes. It consists of 6403 trace files with a total of $46,537,033$ I/O events. Since the LLNL trace is collected at the file system level, any requests not related to metadata operations, such as read, write and execution, are filtered out. Table 3.4 manifests the remaining metadata operations in the LLNL trace. These metadata operations are further classified into two categories: metadata read and

Table 3.4: List of operations obtained by *strace* in LLNL trace collection

Name	**Count**	**Description**
access	16	check user's access permissions
close	111,215	close a file descriptor
fstat64	81,663	retrieve file status
ftruncate64	198	truncate a file to a specified length
open	327,990	open or create a file
stat64	59,892	display file status
statfs	980	display file system status
unlink	8	delete a name and possibly the file it refers to

metadata write before fed into the simulations discussed in Section 3.6.2 and Section 3.6.3. Operations such as *access*, and *stat* fall into the metadata read group, while $ftruncate64$ and *unlink* belong to the metadata write group since they need to modify the attributes

of the file. However, the classification of *open* and *close* is not straight forward. An *open* operation cannot be simply classified as metadata read since it may create files according to its semantics in UNIX. Similarly, a *close* operation can be classified into both groups since it may or may not incur metadata update operations, depending on whether the file attributes are dirty or not. For *open* requests, the situation is easier since we can look at the parameter and return value of the system call to determine its type. For example, if the parameter is O_RDONLY and the return value is a positive number, then we know for sure that this is a metadata read operation. For *close*, an eclectic way is that we can always treat it as a metadata write assuming that the *last modify time* field is always updated upon file closure.

3.6.1.2 HP trace

To provide a more comprehensive comparison, we also conduct our simulations on the HP trace [RKS02], a 10-day trace of file system collected on a time-sharing server with a total of 500 GB storage capacity and 207 users. This 10-day trace covers a period in late 2000 from a Thursday to the following Saturday. It contains 97.4 million file system requests and approximately three quarters of them are metadata requests. Since these traces are relatively old, we scale up the workload collected in this environment to better emulate the projected more intensive workload in a petabyte storage system. We divide each daily trace collected from 8:00am to 16:00pm, which were usually the busiest period during a day, into four fragments, with each fragment containing two hours of I/O accesses. The time

stamps of all events in each fragment are then equally shifted so that this fragment starts at time instant zero. Replaying multiple time-shifted fragments simultaneously increases the I/O arrival rate while keeping a similar histogram of file system calls. In addition, the number of files stored and the number of files actively visited were scaled up proportionally by adding the date fragment number as a prefix to all filenames. We believe that replaying a large number of processed fragments together can emulate the workload of a larger cluster without inadequately break the original access patterns at the file system level. Same as what we did for the LLNL trace, we also filtered out those metadata-irrelevant I/O operations in our simulations.

3.6.2 Simulation framework

A simulation framework was developed to simulate a clustered metadata server (**MDS**) based storage system with the ability to adopt flexible caching/prefetching algorithms. The simulated system consists of 1000 to 8000 compute nodes (clients) and 4 to 256 MDSs. When simulating multiple clients, we basically feed the same trace to all the clients. However, we change all the file names in the trace for each individual client so that the workload is proportionally intensified by increasing the number of clients. For example, a metadata request for file 'A' becomes a request for file 'A1' for client 1, and 'A2' for client 2, and so on. The memory size is set to be 4 GB per MDS and 1 GB per client. All nodes are connected using high speed interconnection with an average network delay of 0.3 ms and a bandwidth

of 1 Gbit/sec under assumption of a standard Gigabit Ethernet environment [IEE05]. The interconnect configuration is the same as shown in Figure 3.1. In such a large, hierarchical, distributed storage system, metadata consistency control on metadata servers as well as the clients becomes a prominent problem for the designers. However, the focus of our current study is the design and evaluation of a novel prefetching algorithm for metadata. To simplify our simulation design, cooperative caching [DWA94], a widely used hierarchical cache design, together with its cache coherence control mechanism, i.e. write-invalidate [AB86], is adopted on the metadata servers in our simulation framework to cope with the consistency issue. The specific cooperative caching algorithm we adopted is N-chance Forwarding, the most advanced solution according to the results presented in [DWA94]. We choose the best cooperative caching solution available for the sake of fair performance comparison. This aims to evaluate the real performance gain from Nexus. From this aspect, it also helps to distinguish the effect of Nexus from that of cooperative caching.

It may also be noticed that the choice of cooperative caching is pragmatic for its relative maturity and simplicity and, as such, it does not necessarily imply that it is the only or best choice for consistency control.

In our simulation framework, the storage system consists of four layers: 1) client local cache, 2) metadata server memory, 3) cooperative cache, and 4) hard disks. When the system receives a metadata request, it first checks its local cache; upon an cache miss, the client sends the request to the corresponding MDS; if the MDS also sees a miss, the MDS looks up the cooperative cache as a last resort before sending the request to disks.

Thus the overall cache hit rate includes three components: client local hit, metadata server memory hit, and cooperative cache hit. Obviously, local hit rate directly reflects the effectiveness of the prefetching algorithm because grouping and prefetching are done on the client site.

If, in the best case, a metadata request is satisfied by the client local cache, referred to as a *local hit*, the response time for that request is estimated as local main memory access latency. Otherwise, if that request is sent to a MDS and satisfied by the server cache, also known as a *server memory hit*, the overhead of network delay is included in the response time. In an even worse case, the server cache does not contain the requested metadata while the cooperative cache does, defined as a *remote client hit*, extra network delay should be considered. In the worst case, when the MDS has to send the request to the disks where the requested metadata resides, i.e., a final cache *miss*, costly disk access overhead will also contribute to the response time.

Prefetching happens when a client sees a local cache miss. In this case the client sends a metadata prefetching request to the corresponding MDS. Upon arrival of that request at the metadata server, the requested metadata along with the entire prefetching group is retrieved by the MDS from its server cache, cooperative cache or hard disk.

3.6.3 Trace-driven simulations

Trace-driven simulations based on aforementioned HP trace and LLNL trace were conducted to compare different caching-prefetching algorithms, including conventional caching algorithms such as LRU (Least Recently Used), LFU (Least Frequently Used) and MRU (Most Recently Used), primitive prefetching algorithms such as First Successor and Last Successor, and state of the art prefetching algorithms such as Noah (Stable Successor), Recent Popularity (also known as Best j-out-of-k), and Probability-Graph Based prefetching (referred to as PG in the rest of this chapter).

Most previous studies use only prediction accuracy to evaluate the prefetching effectiveness. However, this measurement is neither adequate nor sufficient. The ultimate goal of prefetching is to reduce the average response time by absorbing I/O requests before they reach disks. A higher prediction accuracy does not necessarily indicate a higher hit rate nor a lower average response time. The reason is, a conservative prefetching scheme, even with a high prefetching accuracy, might incur little prefetching actions and thus not as beneficial. So, in our experiments, we not only measure the cache hit rate, but also the average response time by integrating a golden disk simulator, DiskSim 3.0 [BG03], into our simulation framework.

We conduct experiments for all the caching/prefetching algorithms mentioned above. For a clear graphic presentation, we remove the results for less representative algorithms, including LFU, MRU (these two are always worse than LRU), First Successor, Last Successor,

Noah, and Recent Popularity, since these algorithms are consistently inferior to PG according to our experimental results and a similar observation made by Pâris in [PAL03]. In addition to these algorithms, Optimal Caching [Knu85], referred to as *OPT* in the rest of this chapter, is simulated as an ideal offline caching algorithm for theoretical comparison purpose. In OPT, the item to be replaced is always the farthest in the future access sequence. Since the prefetching group size for Nexus is set to 2, we have tried both 1 and 2 for this parameter on PG, referred to as PG1 and PG2, respectively, in order to provide a fair comparison. In sum, in this study we will present the results for five caching/prefetching algorithms including Nexus, PG1, PG2, LRU and OPT.

3.6.4 Hit rate Comparison

We have collected the hit rate results for all three levels of caches: client cache, server cache and cooperative cache, as well as the percentage of misses that goes to the server disk, referring to the explanation in 3.6.2.

Figure 3.10 and Figure 3.11 show the hit rate comparison results collected on HP trace and LLNL trace, respectively.

Comparing Figure 3.10(a), 3.10(b) and 3.10(c), it is apparent that with more clients, and thus larger cooperative cache size and smaller per-client server cache size, many requests previously satisfied by the server cache is now caught by the cooperative cache. However, the client local cache hit rate and the overall cache hit rate stay relatively consistent.

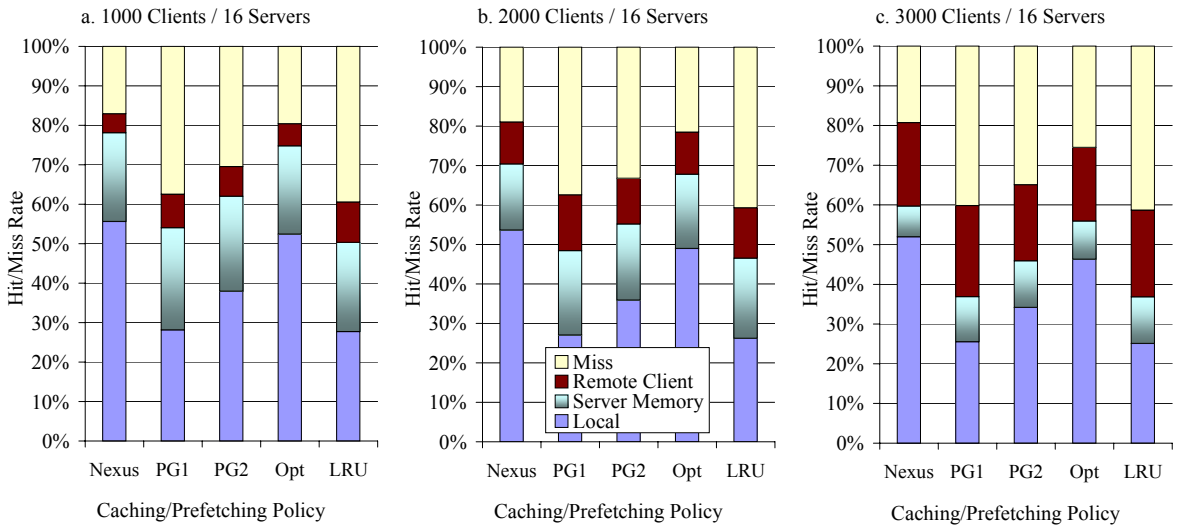


Figure 3.10: HP trace hit rate comparison

In both Figure 3.10 and Figure 3.11, Nexus achieves noticeable better performance on the client local cache hit rate than the other four competitors. For example, Nexus can achieve up to 40% higher local hit rate than that of LRU and PG1. In addition, the fact that PG2 obtains consistent higher client local cache hit rate than PG1 is another implication that advocates the general idea of group prefetching. Based on this reasoning, it seems that a projected PG3 algorithm may potentially outperform PG2 significantly, but its exponential computational complexity prohibited us from further exploring in this direction. Although it is possible to reduce its time and space complexity by more efficient implementations such as regularly filtering out weak links in the relationship graph, however, this topic is out of the scope of our study. It is worth reminding that, Nexus only incurs linear or polynomial computational overhead and thus suits well for group prefetching.

It is surprising to see that Nexus even beats Opt by a small margin (around 3∼10%) in

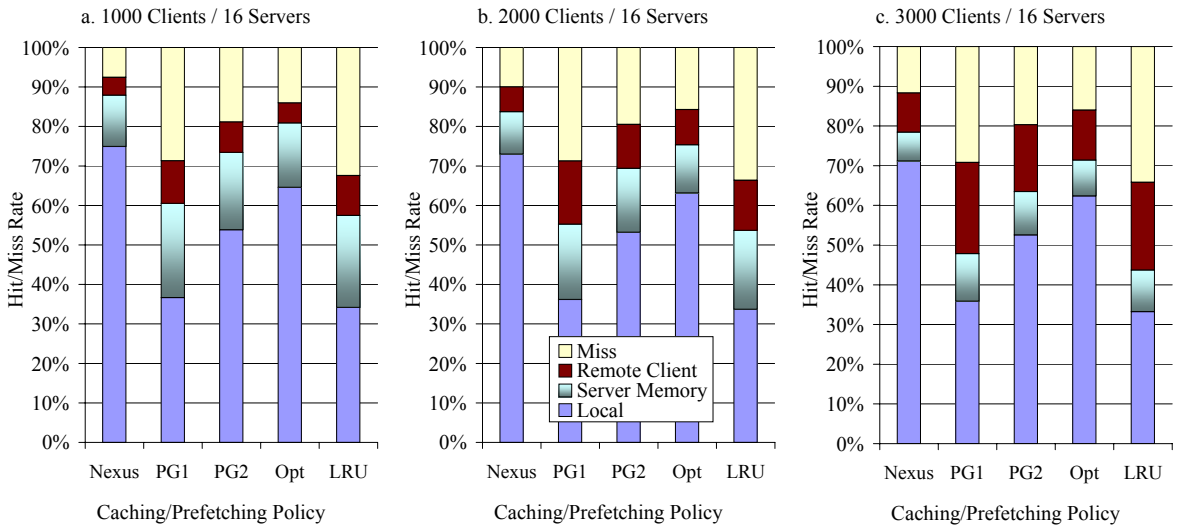


Figure 3.11: LLNL trace hit rate comparison

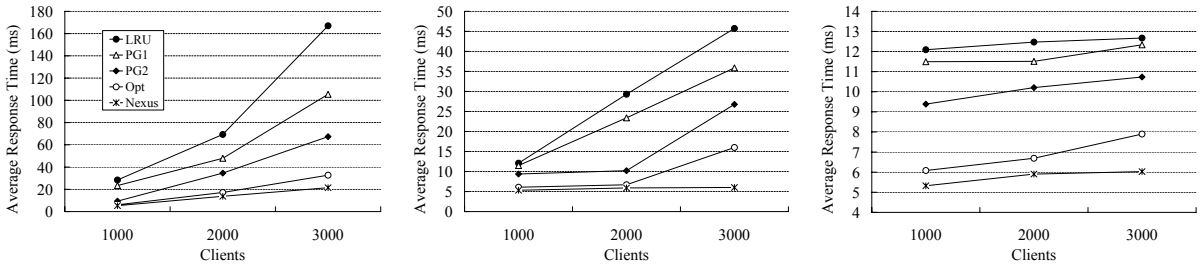
terms of local hit rate, given Opt being the optimal offline caching algorithm with an unrealistic advantage to actually "see" future request sequence before making cache replacement decisions so that . The only limitation of Opt is the lack of prefetching capability compared with Nexus. Consider the situation where object A, B, C and D are always accessed as a group but none of them are currently in the cache, Opt bears four cache misses. However, Nexus will prefetch B, C and D upon a cache miss for A, resulting in one cache miss and three hits.

It may also be worth mentioning that even though Nexus achieves the highest client local cache hit rate, its advantage on overall hit rate is somewhat offset by server cache and cooperative cache. On the other hand, this observation confirms that even the best cooperative caching scheme cannot replace Nexus. At any rate, the overall hit rate does not fully and truly show the merits of Nexus prefetching algorithm. Instead, it is the client

cache hit rate that may exhibit the benefits of Nexus. More importantly, even server cache hit and cooperative cache hit come at the cost of network delay in the range of milliseconds, considerably slower than a local hit which incurs only memory access latency in the range of nanoseconds.

3.6.5 Average Response Time Comparison

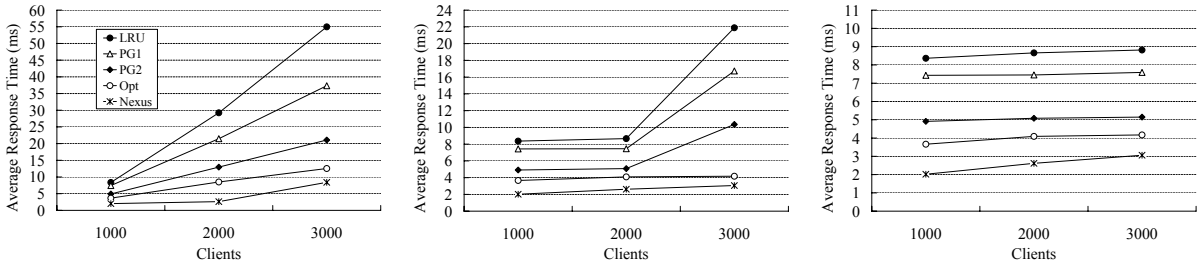
Taking into consideration the possibility that the advantage of prefetching be compromised if too many extra disk accesses are introduced, to accurately measure average response time, we adopted an established disk simulator to incorporate the disk access time in our simulation. The procedure how each single request is serviced is given detailed explanations in 3.6.2. In the experiments, We collect the results for both HP trace and LLNL trace and present their results in Figure 3.12 and Figure 3.13, respectively.



(a) 16 Servers (b) 32 Servers (c) 64 Servers

Figure 3.12: Comparisons of HP Average Response Time per Metadata Request

Apparently, Nexus algorithm excels in all cases in Figure 3.12(a) and Figure 3.13(a). With 16 servers, increasing number of clients from 1000, 2000 to 3000 results in considerable



(a) 16 Servers (b) 32 Servers (c) 64 Servers

Figure 3.13: Comparisons of LLNL Average Response Time per Metadata Request

longer average response time for all algorithms. In contrast, with 32 servers, the average response time for Nexus in Figure 3.12(b) and that of Nexus and Opt in Figure 3.13(b) stays nearly constant while others increase significantly. Furthermore, in Figure 3.12(c) and Figure 3.13(c), the average response time for all algorithms seem to stay little changed. Based on these observations, it seems that individual algorithms exhibit different degrees of "sensitivity" to increasingly intensive workloads. More specifically, systems running Nexus or Opt algorithm are less likely to be saturated under the same workload.

The advantage of Nexus comes from two aspects. First of all, as shown in Figure 3.10 and Figure 3.11, the local hit rate and overall hit rate of Nexus are higher than others. In addition, the computational overhead of this algorithm is kept minimal. Given these advantages, even in cases where the workload stress is relatively high (see Figure 3.12(a) and Figure 3.13(a)), Nexus shows moderate increase of average response time, in contrast to the much more dramatic increase exhibited by other algorithms.

3.6.6 Network Bandwidth Consumption overhead for Nexus

In this section we investigate the network bandwidth consumption overhead introduced by Nexus's aggressive metadata prefetching. In our simulation, the network bandwidth consumption is not measured directly as the percentage of bandwidth used. Instead, it is measured as the total number of metadata requests transferred—including both normal metadata fetching requests and metadata prefetching request—over the interconnect network. The baseline network bandwidth consumption is obtained using LRU without prefetching as the cache management policy. The overhead introduced by Nexus prefetching is calculated as the percentage of extra number of metadata requests to the baseline metadata request number. The corresponding results are shown in Figure 3.14. Comparing these results with those presented in Figure 3.12 and Figure 3.13, a conclusion can be drawn that Nexus uses about 20% more network bandwidth than LRU to reduce the average responde time by 75% to 80%.

3.6.7 Impact of consistency control

The study on the impact of consistency control on the algorithm is also carried out on the HP trace and the LLNL trace. As the results for LLNL trace and HP trace are similar, here we only show the average response time comparison results collected on the HP trace, as in Figure 3.15.

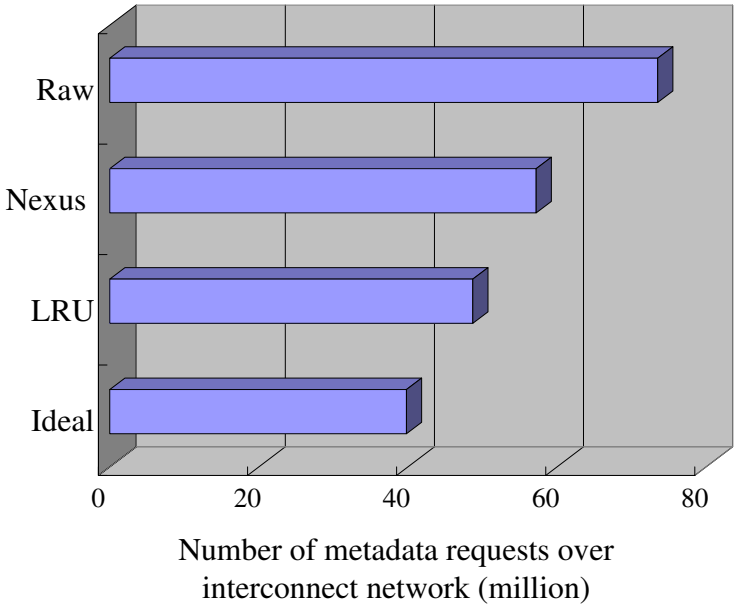


Figure 3.14: Network bandwidth consumption overhead for Nexus. The "Raw" number is obtained with no cache effects, meaning the original number of metadata requests issued by the clients; while the "Ideal" number is obtained with infinite cache, listed here as the theoretical upper bound; "LRU" number is obtained based on LRU cache replacement policy without prefetching; finally, "Nexus" number is obtained based on Nexus prefetching with LRU replacement caching policy.

These results indicate that the average response time was not noticeably affected by the consistency control, within a range of only 5∼10%. In other words, consistency control does not entangle Nexus very much. A possible explanation is that the characteristic of the metadata workloads in this application are either read-only or write-once. In a write intensive workload, the impact of consistency control may become more noticeable. Regarding to the applicability of Nexus in a practical system, similar to other prefecthing/caching algorithms,

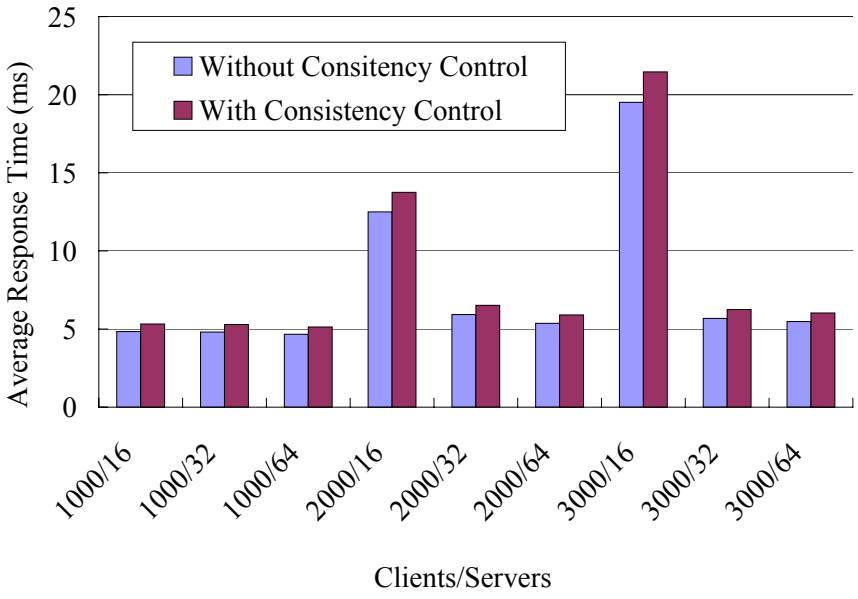


Figure 3.15: Impact of consistency control

our scheme works better for read dominant applications than write dominant applications in order to avoid excessive overhead incurred by the consistent control policy.

3.6.8 Scalability study

In a multi-client multi-MDS storage environment, the system scalability is an important factor directly related to the aggregated system performance. We studied the scalability of the metadata servers equipped with Nexus prefetching algorithm by simulating large numbers of clients and servers. Our evaluation methodology is that keeping constant number of metadata servers, we increase the number of clients and measured the corresponding system throughput, defined by the aggregate number of metadata I/Os serviced per second by the

metadata servers. The results in Figure 3.16 show that, given 4 servers, the throughput

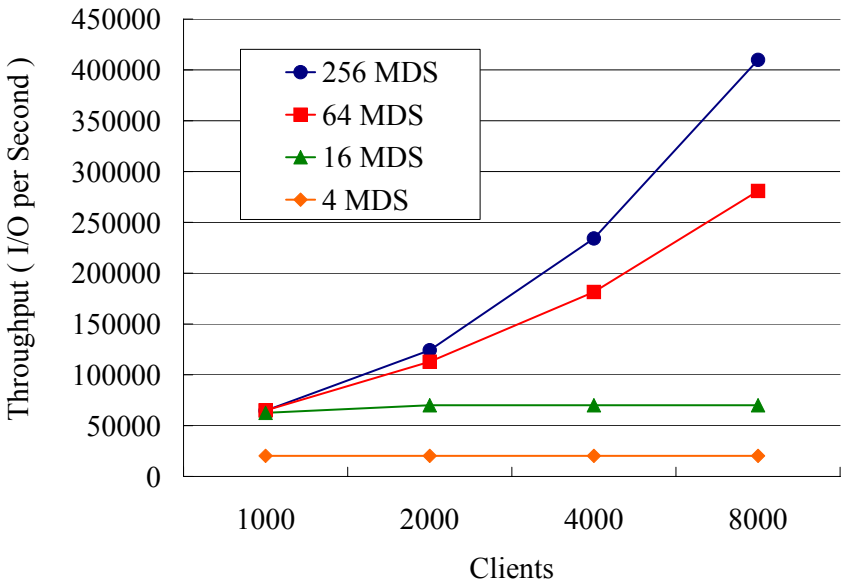


Figure 3.16: Scalability study using HP trace

does not significantly increase while the number of clients increase from 1000 to 8000, as the system is already saturated by 1000 clients at the first place. Prefetching simply can not help when the system is overloaded. In the 16-server case, the throughput increases approximately 6% when the number of clients increase from 1000 to 2000, after that it stops growing since the system became saturated. With 64 or 256 servers, the system throughput scales up almost proportionally with the number of clients, indicating near optimal scalability of the system. As an example, in the 256-server case, the throughput grows from about 6.5×10^4 I/O per second with 1000 clients to about 4.1×10^5 with 8000 clients, more than 6 times increase is achieved.

There are three major factors that contributes to its scalability. First, Nexus algorithm

is totally distributed to clients nodes, there is no central control in our design. System scalability is given serious consideration at the time of Nexus algorithm design. Second, Nexus algorithm runs on client site. That means increased number of clients also provide additional computation power for this algorithm. Third, there is no inter-client communication involved, eradicating the most prominent factor that limits the scalability in many distributed systems.

3.7 Summary

We introduced Nexus, a novel weighted-graph-based prefetching algorithm specifically designed for clustered metadata servers. Aiming at the emerging MDS-cluster-based storage system architecture and exploiting the characteristic of metadata access, our prefetching algorithm distinguishes itself in the following aspects.

- Nexus exploits the ability to look ahead farther than the immediate successor to make wiser predictions. Sensitivity study shows that the best performance gain is achieved when the look-ahead history window size is set to 5.

- Based on the wiser prediction decision, aggressive prefetching is adopted in our Nexus prefetching algorithm to take advantage of the relatively small metadata size. Our study shows that prefetching 2 as a group upon each cache miss is optimal under the two particular traces studied. Conservative prefetching lose the chance to maximize the advantage of prefetching, and too aggressive but not so accurate prefetching might

hurt the overall performance by introducing extra burden to the disk and polluting the cache.

- The relationship strengths of the successors are differentiated in our relationship graph by assigning variant edge weights. Four approaches for edge weight assignment were studied in our sensitivity study. The results show that the linear decremental assignment approach represents the most accurate strength for the relationships.

- In addition to server-oriented grouping, we also explored client-oriented grouping as a way to capture better metadata access locality by differentiating between the sources of the metadata requests. Sensitivity study results show the latter approach's consistent performance gain over the former approach, confirming our assumption.

Other than focusing on the prefetching accuracy — an indirect performance measurement, we pay our attentions to the more direct performance goal — cache hit rate improvement and average response time reduction. Simulation results show remarkable performance gains on both hit rate and average response time over conventional and state of the art caching/prefetching algorithms.

In this study, we make the following contributions.

- We develop a novel weighted-group-based prefetching algorithm named **Nexus** particularly for metadata accesses, featured in aggressive prefetching while maintaining adequate prefetching accuracy and polynomial runtime overhead. Although there exists group prefetching algorithms for data, the different size distributions and access

characteristics between data and metadata are significant enough to justify a dedicated design for metadata access performance.

- We deploys both direct and indirect successors to better capture access localities and to scrutinize the real successor relationship among interleaved accesses sequence. Hence, Nexus is able to perform aggressive group-based prefetching without compromising accuracy. As a comparison, existing group based prefetching algorithms only consider the immediate successor relationships when building their access graphs. In other words, existing group based prefetching algorithms seem to be "short sighted" when compared with Nexus and thus potentially bear less accuracy.

- Finally, in Nexus we defined a relationship strength to build the access relationship graph for group prefetching. The way we obtain this relationship strength makes Nexus a polynomial time complexity algorithm. While other group-based prefetching algorithm, if adopted and made suitable to achieve the same level of "far sight" as Nexus does, could easily be mired in an exponential computational complexity. Therefore, Nexus distinguishes itself from others by its much lower runtime overhead.

CHAPTER 4

BRIDGING THE GAP BETWEEN PARALLEL FILE SYSTEMS AND LOCAL FILE SYSTEMS: A CASE STUDY WITH PVFS

4.1 Chapter Overview

Parallel I/O plays an increasingly important role in today's data intensive computing applications. While much attention has been paid to parallel read performance, most of this work has focused on the parallel file system, middleware, or application layers, ignoring the potential for improvement through more effective use of local storage. In this chapter, we present the design and implementation of Segment-structured On-disk data Grouping and Prefetching (SOGP), a technique that leverages additional local storage to boost the local data read performance for parallel file systems, especially for those applications with partially overlapped access patterns. Parallel Virtual File System (PVFS) is chosen as an example. Our experiments show that an SOGP-enhanced PVFS prototype system can outperform a traditional Linux-Ext3-based PVFS for many applications and benchmarks, in some tests by as much as 230% in terms of I/O bandwidth.

In this chapter we describe an on-disk grouping and prefetching technique named SOGP to bridge the gap between the local storage system and parallel the file system. The main ideas behind SOGP are to store a copy of data that is often accessed in a more efficient organization by grouping noncontiguous file I/O requests and storing these groups (called *segments*) on a local disk partition, and to use this more efficient organization to improve the performance of prefetching at the local storage level, better catering to the needs of parallel file system. By using several synthetic parallel I/O benchmarks, we see that our SOGP-enhanced PVFS scheme outperforms an EXT3-based PVFS by 39% to 230% in terms of aggregate I/O bandwidth in a testbed cluster system.

4.2 Motivation

Recent years have seen growing research activities in various parallel file systems, such as Lustre [Lus04], IBM's GPFS [SH02], Ceph [WBM06a], the Panasas PanFS File System [NSM04], and PVFS [CLR00]. In many cases, parallel file systems use a local file system or object store to serve as a local data repository. Because of the interfaces used to access these local resources, local storage systems are unaware of the behavior of high-level parallel applications that talk directly to parallel file systems. Likewise, because of interface limitations the parallel file system itself is not aware of the underlying local storage organization and operation. In other words, there is an information gap between local file system and parallel file system, and as a result access locality from applications often gets lost. Specifically, the local stor-

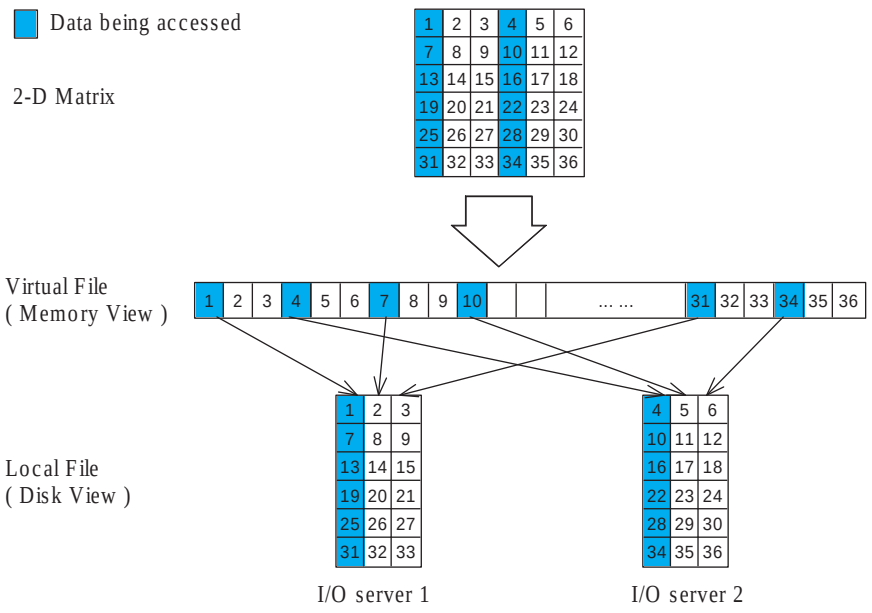


Figure 4.1: Strided Access Pattern

age system can only see accesses to separate pieces of large parallel files. Emerging Object Storage Device (OSD) interfaces, used by parallel file systems [Lus04, WBM06a, NSM04], do appear to present a more appropriate interface for local storage resources, but at this time the OSD interface does not address this knowledge gap.

As an example, if a large matrix is stored on I/O servers in a row-major pattern, while a parallel program needs to conduct column-based processing of this matrix, then requests become noncontiguous at the local storage system level (Figure 4.1). This results in a non-sequential access pattern and reduces the chances of prefetching being appropriately applied. If a sequence of operations, such as visualizing the dataset from multiple viewpoints, will perform column-based operations on the matrix, then a similar pattern of noncontiguous accesses will be repeated each time the matrix is accessed. If we could reorganize the on-disk

data on the fly such that the future accesses become sequential accesses, or keep a copy of the data in this more optimal organization, we could significantly improve the observed local storage bandwidth, and we could do so without changes to the rest of the I/O system.

We note that *partially overlapped accesses* are becoming more common in many emerging scientific and engineering applications [Bry07]: these applications access the same data regions more than once during their execution. Although similar noncontiguous patterns were reported in HPC community one decade ago [NKP96], parallel file system and local file system architectures have not been focused on addressing these patterns. Examples of this pattern of access are common in scientific visualization, real time physical-based rendering, and Geographic Information Systems(GIS) applications. In these applications, when a user zooms in or out or changes viewpoint, some foci regions remain in the display window, but new regions are often accessed to be displayed in the new view (Figure 4.2). When the data is huge and cannot be held in the memory, such as in computational science visualization applications [AMC07], the application is typically re-reading many of the same regions for every new view.

Researchers have pursued two approaches to address the information gap between parallel file systems and local storage systems for better performance: application-directed prefetching hints [CFK96] and language and compiler techniques that automatically insert speculative I/O access statements when compiling application codes [LM99,MDK96,KSK08]. Except for MPI-IO hints, prefetching hints provided by applications are typically limited to specific compilers and thus are not portable. In addition, any application that would ben-

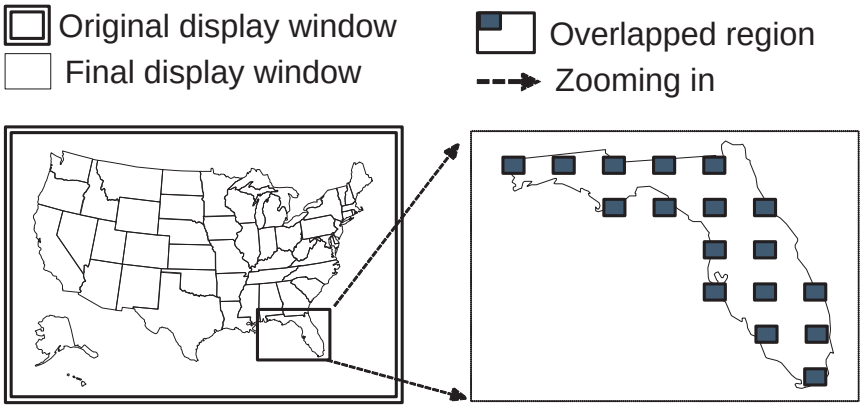


Figure 4.2: When a user use a visualization software to zooms in on a particular region, data in that region is reused. If data must be re-read, the accesses will overlap.

efit from this technique needs to be re-written to accommodate the new feature. Compiler inserted prefetching seems to be more promising. Kotz [KE91] proposed a prefetching technique for parallel file systems as well, based on access patterns studied decades ago. While we believe that this approach is still relevant, we do not believe that prefetching alone is adequate to address the challenges of these applications.

4.3 Related Work

Gokhan Memik *et al.* [MKL06] proposes a new programming model called Multicollective I/O that tries to access a group of unique files with only one MPI-IO alike request. Conceptually, it expands the collective I/O to allow data from multiple files to be requested in a single I/O request, in contrast to allowing only multiple segments for a single file being specified together. In order to do that, the authors propose two different heuristics to detect the access

pattern. However, their work differs from ours since it was done at the MPI-IO library level other than parallel or local file systems. Also, they discover only the inter-file access localities but not intra-file access localities.

Ding *et.al.* propose DiskSeen [DJC07b] and a buffer cache management scheme [DJC07a] to exploit on-disk locality for better prefetching and caching performance. However, their work look at the problem from general applications' angle and their approach is implemented in the kernel, while we are optimizing specifically for parallel scientific and engineering applications and our approach in implemented in a parallel file system at user level. Since we look at similar problems at different angles and implement our approaches at different level, our work should be orthogonal to their approaches.

Kotz *et.al.* proposed several techniques to improve parallel I/O performance including disk-directed I/O techniques [Kot97] and practical prefetching techniques [KE91]. The first work proposes a new technique, disk-directed I/O to allow the disk servers to determine the flow of data for maximum performance by issuing large data requests. The second work developed a local pattern predictor and a global pattern predictor to catch the I/O access patterns in parallel file systems. Our work differs from both of these in that we use a combination of grouping and prefetching to aggregate I/O into large requests and to overlap computation and I/O.

In addition, A number of research projects exist in the areas of parallel I/O and parallel file systems, such as PPFS [JCE95] and PIOUS [MS96]. PPFS offers runtime/adaptive optimizations, such as adaptive caching and prefetching, but does not use segment based

on-disk grouping to maximize disk bandwidth utilization. PIOUS focuses on I/O from the viewpoint of transactions, not from that of scientific computing. In addition, these parallel file systems, as well as I/O optimizations on them, are mostly research prototypes, while our work is done on PVFS, a production-ready parallel file system widely used on Linux clusters.

4.4 SOGP Design And Implementation

In this section, we present the design and implementation of a segment structured on-disk grouping and prefetching technique to improve I/O system performance for parallel file server based parallel file systems. The system works as a cache, so the original data format on local storage is preserved. At this time all metadata for SOGP is stored in memory for performance reasons, so a node failure will cause all the grouping information and on-disk segment cache to be lost, but all data will still be present as stored by the parallel file system, so the reliability of the parallel file system is unaffected by our enhancements. In our current implementation, we choose PVFS as our development and test platform.

4.4.1 SOGP Architecture

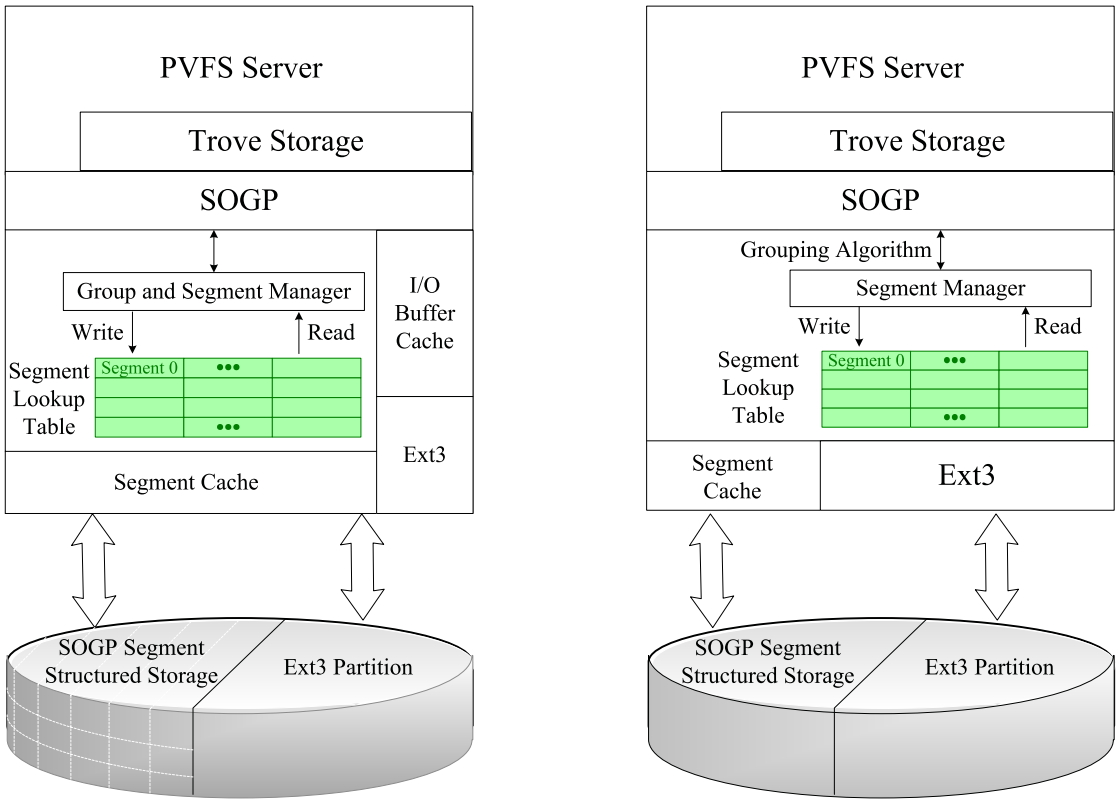
SOGP consists of the following major components: a segment-structured disk storage subsystem, an in-memory segment lookup table, a locality-oriented grouping algorithm and an in-memory segment cache. The disk storage subsystem is adopted to mainly implement seg-

ment I/O, which will be elaborated in Section 4.4.3. The in-memory segment lookup table is a data structure to index and manage all in-memory and disk segments in SOGP.

SOGP design can be implemented either at OS kernel level or user space, as shown in Figure 4.3(a) and Figure 4.3(b) respectively. In former case of Figure 4.3(a), SOGP is actually composed of two parts, a user-level interface to PVFS2 and a kernel module which gets the actual works done (file/block access locality identification, cache management, storage management, etc.). The advantages of the kernel implementation lie in several facts: SOGP could possibly corporate better with the I/O buffer cache; it knows more information about the underlying hardware, and thus might better tune its internal parameters for optimization purposes. The problems of this approach is that it has limited portability and customization room. In addition, since PVFS functionality are implemented purely in user level, and SOGP is designed for PVFS, a user level implementation of SOGP seems more appropriate. Consequently, as an initial trail, we implement SOGP as a user-level component that is integrated into PVFS as a read-only cache, shown in Figure 4.3(b). The SOGP components are detailed in next section.

4.4.2 Augmenting PVFS with SOGP

Central to the implementation of SOGP, how to interact with both PVFS server module and underlying local file system is crucial. According to the design of PVFS, the storage management module is named *trove*. At the time of this writing, the only implementation



(a) SOGP Kernel Implementation

(b) SOGP User-level Implementation

Figure 4.3: SOGP architecture

version of trove is called *DBPF* (*i.e.*, DataBase Plus File). This version uses Berkeley DB for metadata storage and files on a local file system (e.g. ext3) for storing file data. We modified the DBPF code so that SOGP is able to take over the requests dispatched to DBPF. DBPF employs a set of file service functions to handle file I/O requests, among which the most representative ones are *dbpf_bstream_read_list* and *dbpf_bstream_write_list*. These functions are main entries for read/write operations initiated by the upper layers of PVFS. SOGP hooks into these service functions to intercept the requests and process them before they reach the local file system. SOGP monitors the requests received by DBPF and transforms

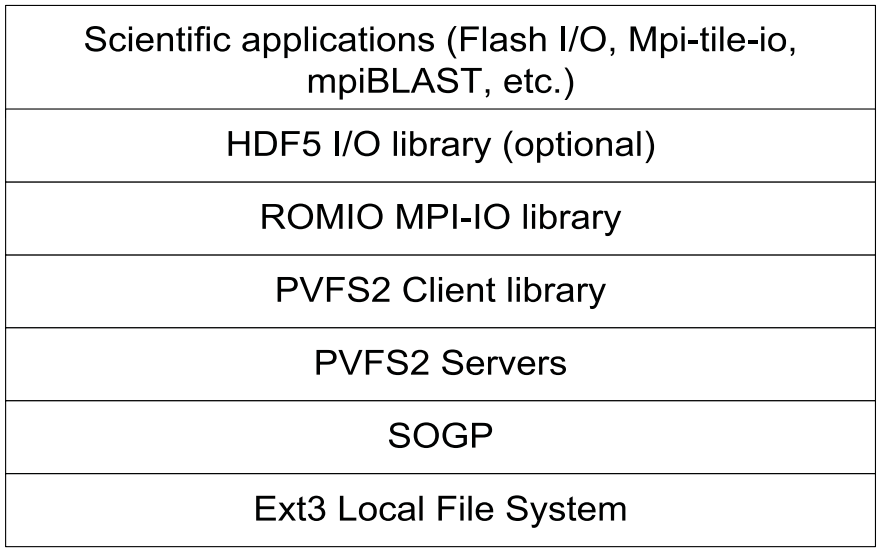


Figure 4.4: PVFS/SOGP software architecture

the requests into its own segment format. For reads, if the segment is cached in memory in SOGP, then the request is satisfied immediately. Otherwise SOGP checks its in-memory segment lookup table to see if the segment is cached on the raw disk partition. If the data is stored in segment form on disk, SOGP uses the POSIX *read/write* system calls to access the raw disk partition as a device special file. If the request is neither cached nor resident on disk, the request is handed over back to DBPF for service. A software architecture of an SOGP-enhanced PVFS2 is shown in Figure 4.4 (Applications and higher layer libraries are also shown for clarity). More details on our I/O interception implementation are provided in Section 4.4.4.

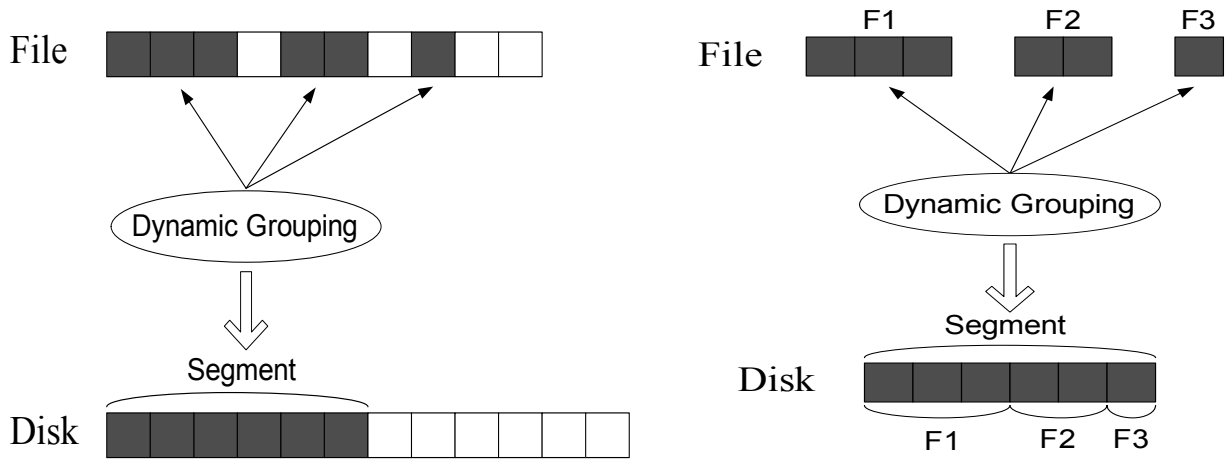
4.4.3 Segment I/O in SOGP

A new I/O technique called *segment I/O* is invented to bridge the mapping gap between logical file layout and physical disk layout by facilitating the large-only disk I/O operations.Disk segment is the atomic unit that contains a large chunk of data in the segment-structured storage subsystem in SOGP. Future references to any files stored in SOGP result in one or more disk segment accesses to enforce a large-only I/O fashion. The compact I/O employs the SOGP algorithm to form groups at runtime. Existing parallel I/O techniques such as data sieving, collective I/O, list I/O, HDF5 and NetCDF optimize the I/O performance at the file system level, library level or application level in the parallel storage system software layer hierarchy. The block level optimization becomes an oversight while appears to be ever important. SOGP implements a compact segment I/O technique that works at both file system level and block level to best improve the performance. In effect, SOGP can be viewed as a complementary scheme to higher level aforementioned I/O solutions. The highlights of the compact segment I/O are listed in the following aspects.

- Small PVFS files accessed by multiple processes are grouped together into segments as the basic access unit, exploiting the inter-file access locality.

- Hot portions of large PVFS files accessed by multiple processes are grouped together into segments, exploiting the intra-file access locality.

Figure 4.5(a) and Figure 4.5(b) illustrate two common scenarios — accessing multiple large file portions and small files by the compact segment I/O respectively.



(a) Example 1: Multiple file regions

(b) Examples 2: Multiple small files

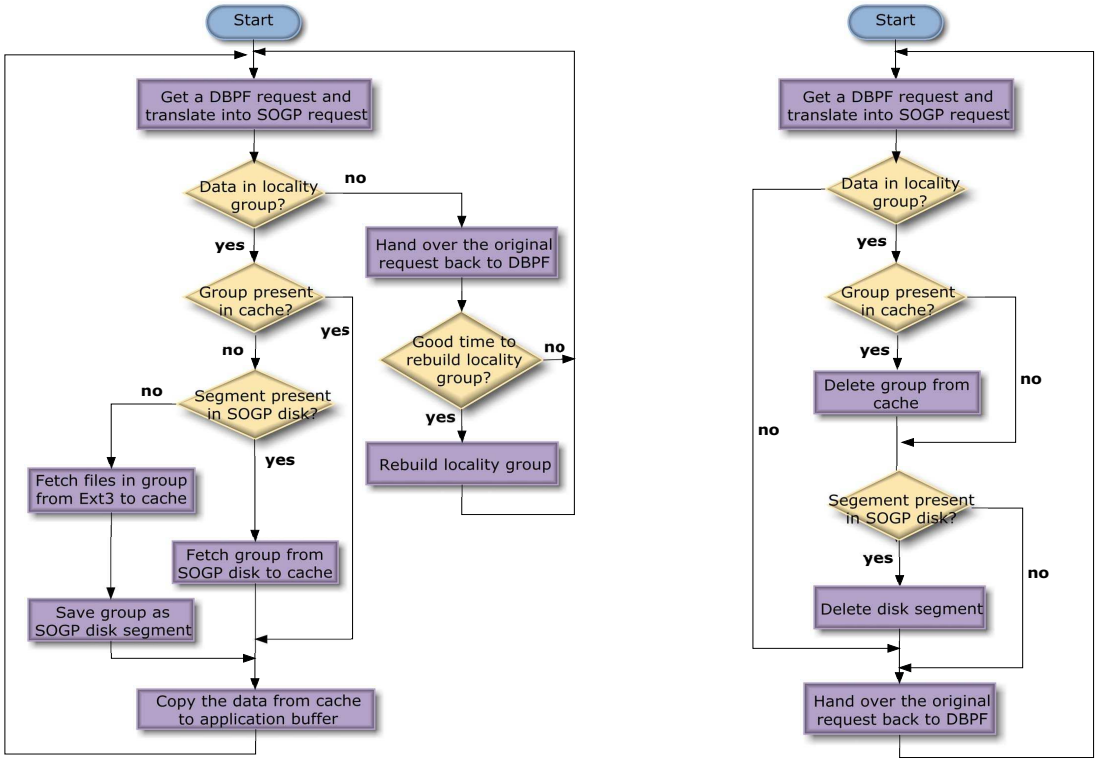
Figure 4.5: Compact segment I/O examples

4.4.4 SOGP Data Flow

4.4.4.1 Read handling

When SOGP receives a read request, it works in the following steps (See Figure 4.6(a)). Note that in Steps 4 and 5, the read operation fetches not only the requested data, but also other data from the same or related files in the same segment group. In other words, prefetching is implicitly performed in these steps.

1. Receive a DBPF read request and translate the request into SOGP format, i.e., a SOGP read request.



(a) Read request handling

(b) Write request handling

Figure 4.6: SOGP read/write data flow

2. Determine if this file belongs to any groups identified by its dynamic grouping algorithm. If so, continue to next step; otherwise, hand over the original request back to DBPF and then rebuild the locality group when necessary.

3. Check the in-memory lookup table to determine if this group is resident in SOGP in-memory cache or on SOGP storage. If in memory, satisfy the request immediately by copying data from the cache.

4. If the data is on SOGP storage, perform raw partition read to retrieve data into the segment cache and return the data.

5. Otherwise, fetch the data from the local file system to the segment cache. Since the requested file belong to a segment group which is not on SOGP, we also save the group on SOGP as a segment for future reuse before the request is returned to the user.

4.4.4.2 Write handling

In our prototype implementation, SOGP does not perform optimizations on write requests. We employ write-invalidation as the segment cache consistency policy, so if write data is found to be on one of SOGP segments, then the entire segment is invalidated from both the disk and the memory (if present). After clearing any old data out of SOGP, write requests are handed over back to DBPF. The detailed steps for writes are described in Figure 4.6(b).

4.4.5 Locality-based Grouping and Prefetching

The purpose of SOGP is to group noncontiguous file I/O requests into segments *at runtime*, helping to improve data locality for the parallel file system and higher level applications. Segments are stored as physically continuous chunks on raw disks or partitions. These segments are our atomic storage unit to read, prefetch, write, and invalidate. The segment size is 16 MB in our current design trying to match the disk cylinder group size [SSS99]. By co-locating related data in the same cylinder group, we reduce the number of small I/Os and help maximize read performance. The success of the technique is dependent on our ability to detect and exploit locality in parallel file access. The design goal of SOGP is to

reduce number of I/Os by exploiting the access locality among files. In our current prototype system, SOGP functions as read-only cache in front of the local file system. Since we are using a modular design, the functionality of SOGP can be easily extended in the future when desired.

4.4.5.1 Two levels of Grouping in SOGP

We notice that there are two levels of temporal and spatial access locality in existing parallel applications: inter-file access locality and intra-file access locality.

Inter-file level access locality Small file access pattern bears access locality almost in all kinds of applications. For example, when we compile our program, the same set of header files may be included many times by difference source files. In a similar case, when we write a paper using Latex, every time when the work is compiled, the same set of packages are included, and therefore the same set of files are accessed. This implies grouping multiple small files together could facilitate large I/O operations. In this way, the entire group of files are fetched when any one of the group members is accessed. We observe such pattern exists in current scientific computing [MR02, WK03] as well.

Intra-file level access locality Many compute nodes may concurrently read the same large data file, e.g., a 3-D object database, but in different portions. Researchers have noted several representative access patterns existing in today's scientific computing applications,

such as simple strided, nested strided, random strided, sequential, segmented, tiled, and unstructured mesh accesses [Sho03]. For example, Figure 4.1 illustrates a nested strided access pattern resulting from column based access to a 2-D matrix. Figure 4.2 shows the effect of zooming in on an image. In these examples, the innate access locality between adjacent column elements disappears at both the file level and the local storage level. The problem is that parallel file systems may understand these access patterns but do not have control of the disk data organization; while local file systems do have control of the data layout, but they lack knowledge of the higher level parallel I/O access patterns. SOGP works around the knowledge gap at the local storage level by speculating on the parallel I/O access pattern and controlling the local disk layout to better suit this pattern, bridging the gap between parallel file systems and local file systems.

4.4.6 Grouping algorithm considerations

Unique to the aforementioned data grouping algorithms, our locality-based grouping algorithm is capable of discovering group access locality at both file level and disk level. To achieve this goal, in SOGP design, both small files (inter-file access locality) and large file portions (intra-file access locality) are treated the same way as individual data units of SOGP segments. These data units are then represented as nodes in a graph, which maintains the group access relationship among the individual data units. It should be noted that our grouping algorithm is not intended for prefetching continuous data in a large file, rather,

it is designed to improve the I/O performance for accessing small files and noncontinuous portions of a large file by grouping them together for future prefetching.

To develop a grouping algorithm for our special purpose, there are several specific issues to bear in mind.

Accuracy First of all, the grouping algorithm has to be accurate. Data prefetching based on inaccurate grouping information can easily introduce overhead significant enough to balance out its advantage. The possible overhead of mis-grouping is at least two fold: waste of disk bandwidth and pollution of the SOGP group prefetching cache. As a result, accuracy is always the first priority through our grouping algorithm design.

Efficiency Secondly, the grouping algorithm must be efficient. We are designing an online grouping and prefetching algorithm rather than an off-line one. An online system has to be efficient in terms of CPU cycles for practical use. Even if the prefetching algorithm turns out to be highly accurate, an inefficient algorithm will not justify for an online usage.

Adaptivity Thirdly, this grouping algorithm must be adaptive to the workload. The reason is that I/O requests do not always arrive in constant and regular patterns over time. Therefore, the grouping algorithm have to adapt well to the workload changes. In addition, the capability of adaptiveness has to considered at design stage.

4.4.7 Grouping algorithm and its complexity

We found that *Probability-based Successor Group Prediction* [ALB02] appears to be a good candidate for our grouping purpose. Unfortunately, it is not a practical solution due to its spatial complexity: it requires unbounded memory to hold the entire online I/O trace in order to calculate the probabilities of one file being a successor of another.

Since we are working on data grouping which requires ideal accuracy, we choose to use Recent Popularity algorithm to build the relationship graph. By adjusting the parameters j and k in best-j-out-of-k algorithm, we can control the accuracy of the prediction algorithm. Once the graph is built, we need to divide the nodes into groups for prefetching. For the purpose of prediction accuracy, we adopt the most strict graph partitioning algorithm — Strongly Connected Component algorithm [CLR01]. To help better understanding our grouping algorithm, the pseudocode of recent popularity algorithm is described in Table 4.1. The description on the well-known Strongly Connected Component algorithm can be found in [CLR01].

Next we will examine the complexity of our two algorithms, Recent Popularity for building the relationship graph and Strongly Connected Component for graph partitioning.

More formally, the problem of Recent Popularity can be rephrased as follows: given a trace T consisting of a sequence of elements, try to build a relationship graph G using best-j-out-of-k algorithm according to T. In order to find out the computational complexity, a naive algorithm to construct graph G is given below.

Table 4.1: Pseudocode for Best-j-out-of-k algorithm

BEST-J-OUT-OF-K-GRAPH (T,j,k)

1 Build a set S containing all the unique elements of T

2 Initiate an empty queue $Q[S_i]$ of fixed length k for each elements of S S_i

3 **for** $i \leftarrow 1$ **to** $|T| - 1$

4 **do** Enqueue $T[i+1]$ to proper queue $Q[S_h]$ such that $S_h = T_i$

5 $G \leftarrow \emptyset$

6 **for** $\forall(m, n)$ such that $S[n]$ appears at least j times in $Q[S_m]$

7 **do** add edge $E(m, n)$ to graph G

8 **return** G

In this algorithm, suppose the size of T is n and the size of S is m ($m \leq n$), we calculate the algorithm complexity step by step. For step 1, the time required to go through the entire sequence of T is $O(n)$. Step 2 requires $O(k \times m)$ time. Step 4 is composed of searching S_h for T_i ($O(m)$ time) and enqueueing T_{i+1} ($O(1)$ time). Since it is repeated $n-1$ times, the total time required by step 3 and 4 is $O(m \times n)$. Step 5 obviously requires constant time ($O(1)$). For step 6, the maximum number of iteration will be $\frac{m \times k}{j}$ and step 7 requires $O(1)$ time. Hence, the total time required for step 6 and 7 is $O(\frac{m \times k}{j})$. Summing up the time required by each step, we get an accumulative time complexity of $O(m \times n)$. Hence, we conclude that this is a polynomial time algorithm.

For the computational complexity of Strongly Connected Component algorithm, an well known algorithm of $O(|V| + |E|)$ complexity is already given based on Depth First Search. The proof can be found in [CLR01]. Hence, we know that this is a linear time algorithm.

Finally, the total cost for building and partitioning the graph is the combinational cost of aforementioned two algorithms, in other words, a polynomial time complexity.

4.4.8 Scheduling grouping

It is critical that grouping should not compete with normal I/O operations. In our design, we choose storage system idle periods to perform data grouping. In order to do this, we modified the block device driver to allow reporting the length of its request queue. SOGP periodically checks this to make sure the queue is empty before sending grouping requests. If an idle period is detected, then the grouping request is sent. In our current implementation, we perform this queue length query once per second so that the overhead of this query is kept at a very low level. In addition, the idle period threshold is set to five seconds.

4.4.9 Discarding prefetched group items

The prefetched items are located together in memory with regular cached items. It is possible that prefetched items get expunged before related requests arrive. In a parallel file system, prefetching and caching can both improve the I/O performance. However, prefetching is

more effective in parallel scientific computing domain [LSK01]. In the typical situation, it would be desirable to have a large space for prefetching. With the help of SOGP, such demand is largely alleviated, because grouped segments are sequentially stored on disk, and, to retrieve them from disk again, large I/Os requesting many adjacent blocks are issued to the disk, fully utilizing the maximum disk bandwidth.

4.5 Evaluation

Our evaluation was performed on two clusters. The first cluster, the Computer Architecture and Storage System (CASS), is a departmental storage cluster servicing multiple research groups at the University of Central Florida. The CASS cluster consists of 16 Dell PowerEdge 1950 nodes. Each node has two dual-core Intel Xeon 2.33 GHz processors, 4 Gbytes of DDR2 533 memory, and two SATA 500 Gbyte hard drives or two SAS 144 Gbyte hard drives. Each node has two Gigabit Ethernet ports, one for management and one for data transfer. These nodes are connected with a Nortel 5510-48T non-blocking 48 port high speed network switch and running the Red Hat Enterprise Linux operating system. PVFS 2.7.0 is installed on each of these nodes with the same configuration. Eight nodes out of the 16 nodes are configured as dedicated PVFS storage nodes and each of them assumes multiple roles: PVFS server and PVFS client. The remaining eight nodes are configured solely as compute nodes. All PVFS files were created with the default 64 KByte strip size, summing up to a 512 KByte stripe across all the eight server nodes. In addition, MPICH2 version 1.0.6p1 is installed

as the MPI library. Unless stated otherwise, the tests on the CASS cluster are performed with 8 PVFS server nodes and 16 PVFS client nodes. (Note that each node has four CPU cores). Each test is repeated five times and the average is presented. The caching effect is deliberately avoided by rebooting the server nodes between runs.

To further test the scalability, we also ran some parallel I/O benchmarks on the Chiba City cluster at Argonne National Laboratory. The Chiba City cluster is a 512 CPU cluster running Linux. The cluster also includes a set of eight storage nodes. Each storage node is an IBM Netfinity 7000 with 500 MHz Xeons, 512 MBytes of RAM, and 300 GBytes of disk. The interconnect for high performance communication is 64-bit Myrinet. All systems in the cluster are on the Myrinet. The software stack is the same as CASS.

4.5.1 Software Configuration

We choose to compare the SOGP solution with an installation using the Ext3 file system, the most popular native file system for Linux-based clusters where PVFS resides. For brevity, in the rest of this chapter, PVFS with Ext3 support is referred as PVFS/Ext3, while PVFS with SOGP as PVFS/SOGP. When performing experiments, we alternated between PVFS/ext3 and PVFS/SOGP. Two independent PVFS "storage spaces" were configured, with the PVFS/SOGP configuration using Ext3 for DBPF data and a separate local disk partition for cached SOGP data. Next we describe our benchmarks and applications in detail and compare the results of both I/O systems.

4.5.2 Benchmarks

We use three popular parallel I/O benchmarks to evaluate the benefit of our design over the traditional PVFS/Ext3 approach. We use mpi-tile-io to simulate visualization application behavior, and we use noncontig and IOR to simulate zooming behavior in various visualization applications.

As far as the I/O intensive parallel application is concerned, the most important thing users would be interested in is the aggregate I/O bandwidth (*I/O bandwidth* in brief for the rest of the chapter) of the entire parallel file system, which is the sum of the I/O bandwidth of all storage nodes. As a result, we choose the I/O bandwidth as the major metric during the evaluation.

Noncontig is a publicly available parallel I/O benchmark from Parallel I/O Benchmarking Consortium [pio]. It is designed for studying I/O performance using various I/O methods, I/O characteristics and noncontiguous I/O cases. This benchmark is capable of testing three I/O characteristics (region size, region count, and region spacing) against two I/O methods (list I/O and collective I/O) in four I/O access cases (contiguous memory contiguous disk, noncontiguous memory contiguous disk, contiguous memory noncontiguous disk, and noncontiguous memory noncontiguous disk).

In real world visualization applications, image zooming is an important and common behavior [WS06]. For a very large dataset, the user might want to see a single picture that

represents the entire data at first. After a quick preview, the user might want to focus on a particular region of interest, i.e., zooming to see more detail. Zooming is simulated by increasing veclen while reducing elmtcount so that the product of them is kept constant.

Mpi-tile-io is another synthetic benchmark from the Parallel I/O Benchmarking Consortium benchmark suite [pio]. It has been widely used in many parallel I/O related studies [WK03, YLP05, WWP03]. The application implements tile access on a two-dimensional dataset, with overlapped data between adjacent tiles. The size of the tiles and the overlap ratio is adjustable. Collective I/O support is optional in this application. We studied both cases with and without collective I/O support in our experiments.

IOR is developed at Lawrence Livermore National Laboratory [IOR]. It is designed for benchmarking parallel file systems using POSIX, MPI-IO, Parallel netCDF, or HDF5 interfaces. To test the scalability of our SOGP design, we run the IOR benchmark on the Chiba City cluster by varying the number of clients from 8 to 256. At the same time, we also turned on the use file view option in IOR and changes the view during runs to keep the total size of working set constant while simulating the partially overlapped access pattern.

4.5.3 Applications

CP2K is a freely available program under GPL licence, written in Fortran 95, to perform atomistic and molecular simulations of solid state, liquid, molecular and biological systems.

It provides a general framework for different methods such as e.g. density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW), and classical pair and many-body potentials. CP2K provides state-of-the-art methods for efficient and accurate atomistic simulations. Although CP2K is a simulation in the eyes of physicist, it is an real world scientific application from the perspective of computer system researchers. We use this application to test its performance on PVFS/SOGP in comparison to that on PVFS/Ext3, to evaluate the performance gain introduced by SOGP in real world scientific applications.

SIESTA (Spanish Initiative for Electronic Simulations with Thousands of Atoms) [SAG02] is both a method and a computational implementation, to perform electronic structure calculations and ab initio MD simulations of molecules and solids. SIESTA uses self-consistent density functional theory (DFT) for the calculation of the electronic structure. In addition to this, it offers options to modify the nuclear variables, such as molecular dynamics simulations, optimization and phonon calculations. It uses a linear combination of atomic orbitals (LCAO) as basis set. There is a choice of direct solver or iterative solver for the eigenvalue problem. The iterative solver scales linear with the number of atoms. Like CP2K, although it may be considered simulation by physics researchers, it is also deemed a good candidate application for computer system performance study. As for the CP2K case, we use it to examine the performance difference between PVFS/SOGP and PVFS/Ext3.

4.5.4 Prefetching Accuracy

In order to better understand the source of benefit — group access locality, we further investigate SOGP application-level buffer cache — group cache behaviors. In our experiments, we enable a small amount of logging code in SOGP to collect the group cache utilization statistics while PVFS/SOGP server is running. The results are then written into a server log file. Figure 4.7 describes where the logging code is inserted. The storage system layer

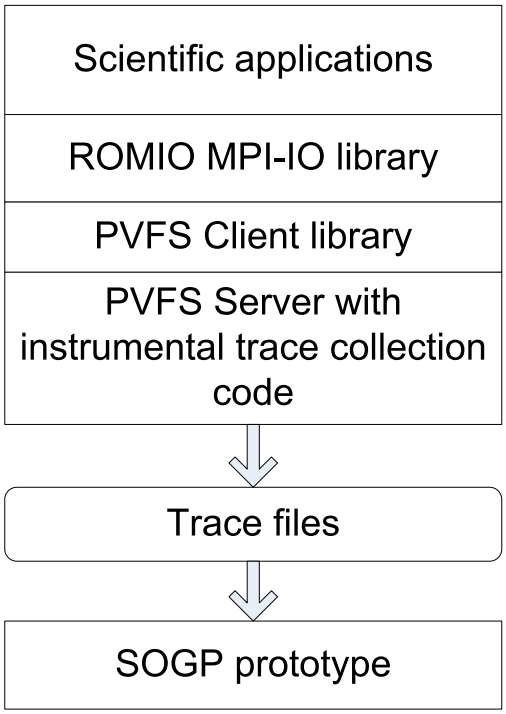


Figure 4.7: Trace Collection for SOGP Prefetching Accuracy

configuration at the time of testing is described in Table 4.2. We ran noncontig benchmark five times with different parameters to simulate the image zooming effect. The aggregated working set is approximately 100 GB. After the test, we extract the data from the server log

Table 4.2: Storage system layers configuration

SOGP segment cache	SOGP partition	Ext3 partition
1 GB	20 GB	180 GB

file, as shown in Table 4.3. There are 52917 segment groups in total built during the test and 98.27% of them are accessed more than once.

Table 4.3: Group access locality analysis

Number of processes	Hits	Misses	Total	Hit rate(%)
4	4090515	654311	4744826	86.21
16	4144802	726278	4871080	85.09
64	4421932	864930	5286862	83.64
256	4652093	1051098	5703191	81.57

4.5.5 I/O Bandwidth

For the noncontig benchmark, which exhibits noncontiguous file accesses in some phases, we expect that PVFS will significantly benefit from SOGP in these phases. For contiguous file access phases, the corresponding disk accesses may still become non-contiguous because of the gap between the file system and the disk, and therefore PVFS can still possibly benefit from the group access feature of SOGP.

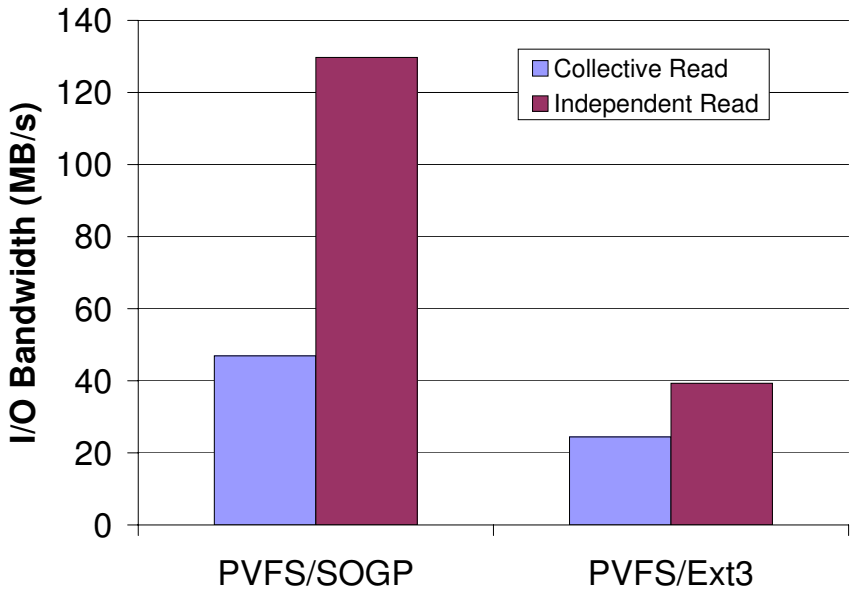


Figure 4.8: noncontig I/O performance comparison

In addition to the performance implications of SOGP, we notice that the results for collective I/O method and non-collective I/O method exhibit some differences. We ran the test on CASS cluster and collected results for both collective and non-collective methods, as shown in Figure 4.8 to allow a side-by-side comparison.

In Figure 4.8, PVFS/SOGP exhibits a read performance gain of 92% to 230% over the PVFS/Ext3 baseline system. These results suggest that, by combining highly related accesses into groups, SOGP can boost the I/O performance dramatically. The simple strided pattern of noncontig benchmark issues I/O accesses to the same data repeatedly, which translates into better group access locality that SOGP is able to take the best advantage of.

We were concerned that these results might result from a particularly good match between the vector length used in the test, which determines the number of regions and region spacing,

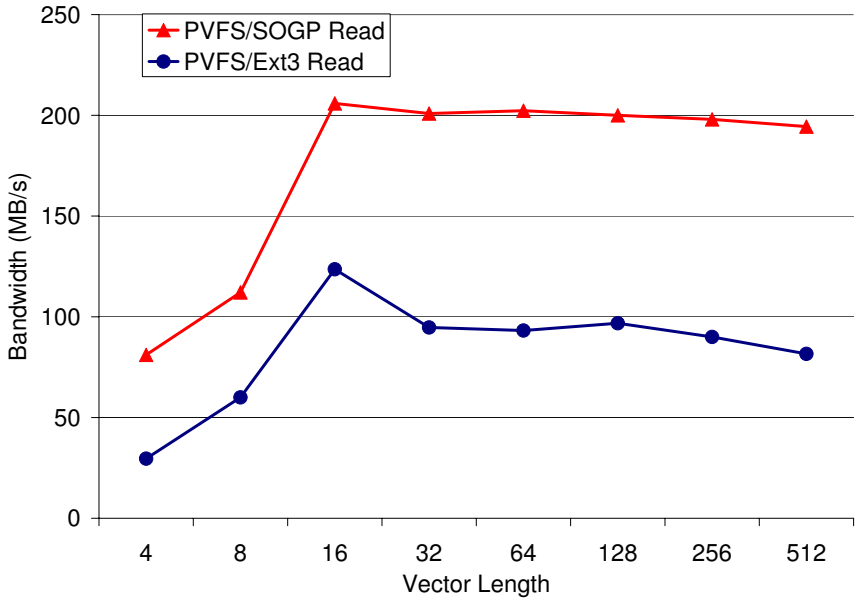


Figure 4.9: Read performance for noncontig, varying vector length

and our SOGP configuration. Figure 4.9 explores the impact of vector length on performance for the range between 4 Kbytes and 512 Kbytes. We see that at very small sizes there is a drop-off in performance, possibly due to the general inefficiency of servicing a very large number of small and noncontiguous regions, but otherwise the performance improvement is consistent.

Figure 4.10 presents the I/O bandwidth results collected on CASS when running mpi-tile-io on PVFS/Ext3 and PVFS/SOGP, respectively. In this test, the total request size was 128 GBytes. To show that the performance impact of collective I/O is orthogonal to that of SOGP, we plot the results for both collective I/O and independent I/O (non-collective) method in this figure. In both cases, PVFS/SOGP exhibits approximately 50% higher I/O bandwidth than PVFS/Ext3. This derives from PVFS/SOGP grouping multiple small I/Os

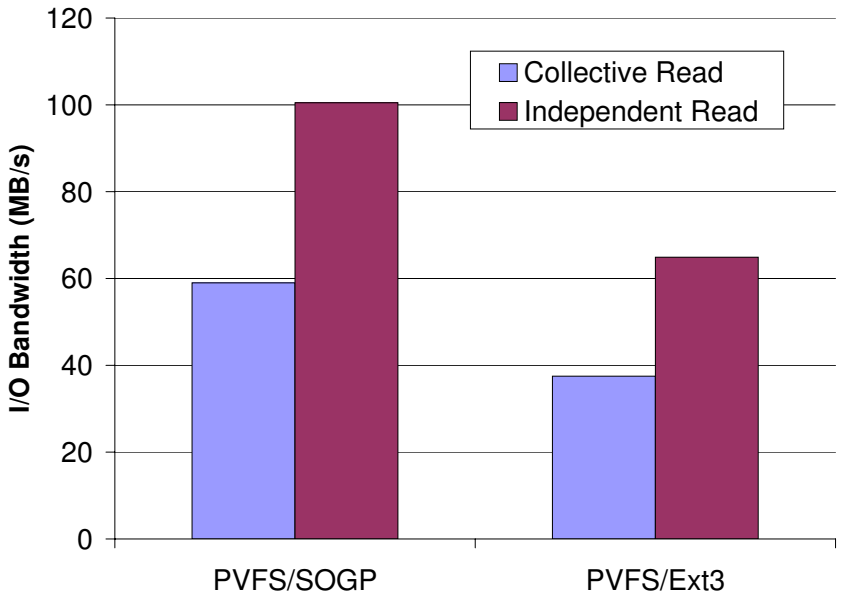


Figure 4.10: I/O bandwidth with mpi-tile-io

into larger ones for read access and prefetching. With this total size, the entire dataset fits into the SOGP partitions on the servers, allowing for better locality of access on reads. A possible reason why the performance gain is not that "conspicuous" lies in that mpi-tile-io, as a read-once/write-once dominant application, does not make good use of the grouping feature of SOGP in a repeated fashion.

4.5.6 Overlapped Access

In this section we analyze the percentage of overlapped access in several applications such as mpi-tile-io. These applications are known to have overlapped accesses. For mpi-tile-io, the tiles to be accessed can be specified by the vertical and horizontal spread of the tile, so it is

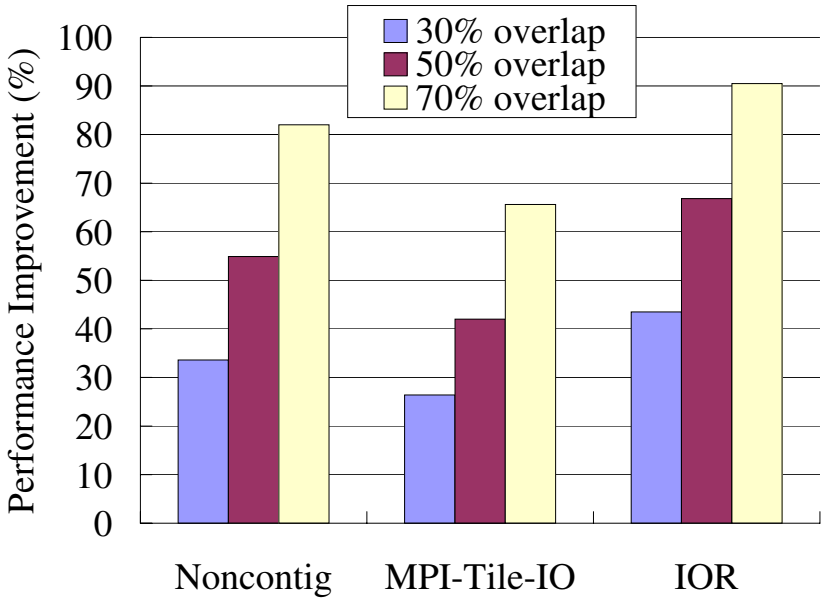


Figure 4.11: Impact of access overlap on I/O aggregate bandwidth

easy to specify to what degree accesses will overlap. Since the resulting I/O accesses are not exactly the same, in many cases they are only partially overlapped. We defined the partially overlapped access as the *number of bytes* accessed more than once. We use this number to the total number of bytes accessed to obtain the percentage of overlapped access.

In Figure 4.11 we present the impact of overlapped access over performance gains of SOGP. As we would expect, the larger the percentage of overlapped accessed region, the larger the benefit of grouping.

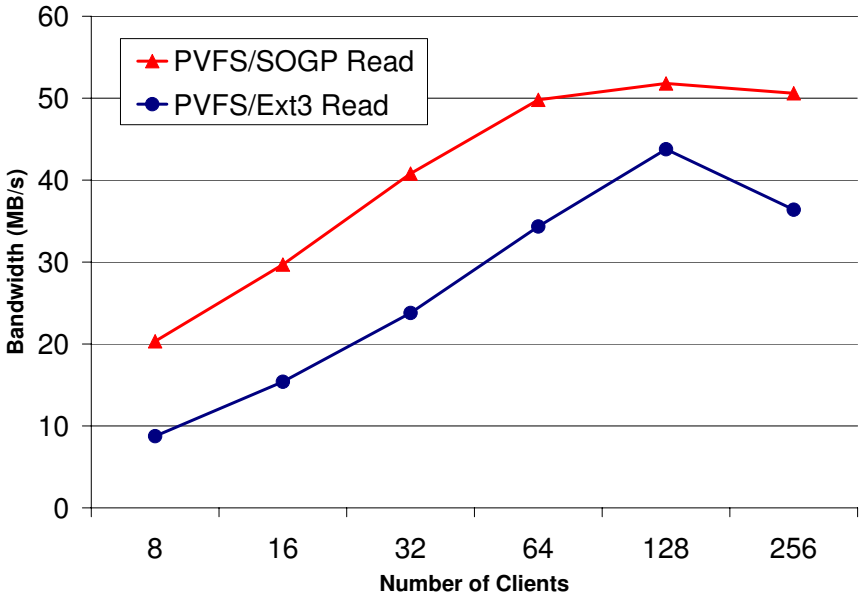


Figure 4.12: IOR benchmark bandwidth comparison

4.5.7 Scalability study using IOR

From the results shown in Figure 4.12, one can observe that both PVFS/SOGP and PVFS/Ext3 show performance gains in proportion to the number of clients (i.e., processes) within a certain range (less than 128). The I/O bandwidth of both systems degrades at 256 clients on this test system, likely because the PVFS servers are saturated. On the other hand, PVFS/SOGP outperforms PVFS/Ext3 by up to 132% in terms of absolute read performance (in unit of I/O bandwidth). Even when the PVFS becomes saturated, PVFS/SOGP performance exceeds that of PVFS/Ext3 by 39%. Another observation is that the benefit gain of PVFS/SOGP over PVFS/Ext3 from group access locality is not decreased by increasing the number of processes that issue the read requests. These results indicate that

for read workloads PVFS/SOGP retains the scalability properties of PVFS while providing significantly higher bandwidth at a given number of clients.

4.5.8 CP2K performance comparison

In this section we test the benefit of PVFS/SOGP over PVFS/Ext3 by running CP2K code on our CASS cluster. We use wall time as the basic to measure the performance of CP2K in both PVFS environments. We perform two kind of tests for CP2K on CASS cluster, one for *fixed problem size* and one of *scaled problem size*. With fixed problem size, the size of total data set accessed by the application does not change with increased number of clients. Under the scaled problem size case, increasing the number of clients will result in larger aggregated data set being accessed. Each simulation cell is composed of a series of unit cells replicated in three Cartesian directions. Furthermore, each unit cell is composed of eight nitrogen atoms in the cubic gauche (cg-N) crystal structure. The shape and size of the simulation cell is denoted by $l \times w \times h$, where l, w and h are integers representing the dimension in the three dimensions of a Cartesian space. We tests four combination of the h, l and w value in our experiments, namely $3, 6 \times 3 \times 3, 6 \times 6 \times 3$ and $3 \times 3 \times 32$.

For these four environments, in figure 4.13, we show the performance speedup of CP2K in accordance with the increased number of clients under fixed problem size. In general, CP2K shows good scaling performance in all experiments. By comparing the speedup of CP2K application on PVFS/SOGP with that on PVFS/Ext3, we can see a notable distinction

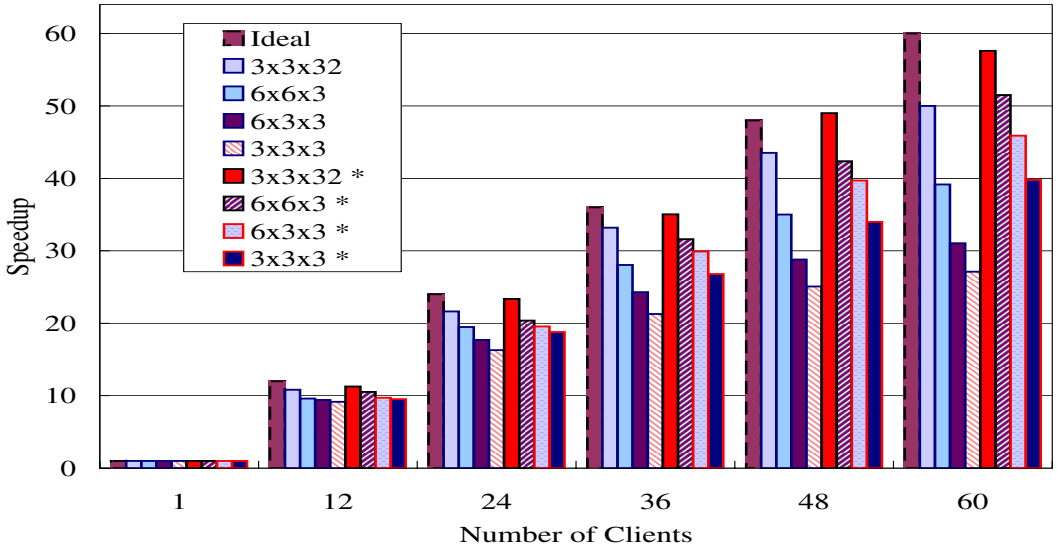


Figure 4.13: CP2K Speedup with Fixed Problem Size. Ideal represents linear speed up, legends with * represent the results for PVFS/SOGP, those without * represent the results for PVFS/Ext3.

between those two groups. For the $3 \times 3 \times 3$ case (meaning the simulation cell is cubic), with 60 clients, CP2K on PVFS/Ext3 presents a speedup of less than 30, showing a scaling efficiency of less than 50%. However, for the same experiment, CP2K on PVFS/SOGP achieves a speedup of 40, showing a scaling efficiency of around 66%. The final performance difference based on these two different platforms is as significant as 47%. For other cases, although the performance difference is not as significant as the aforementioned one, the distinction between the use of PVFS/Ext3 v.s. PVFS/SOGP is typically between 10% and 45%, and thus clearly observable. Another observation is that the more "cubic" the simulation cell is, the better performance gain is achieved by using PVFS/SOGP than using

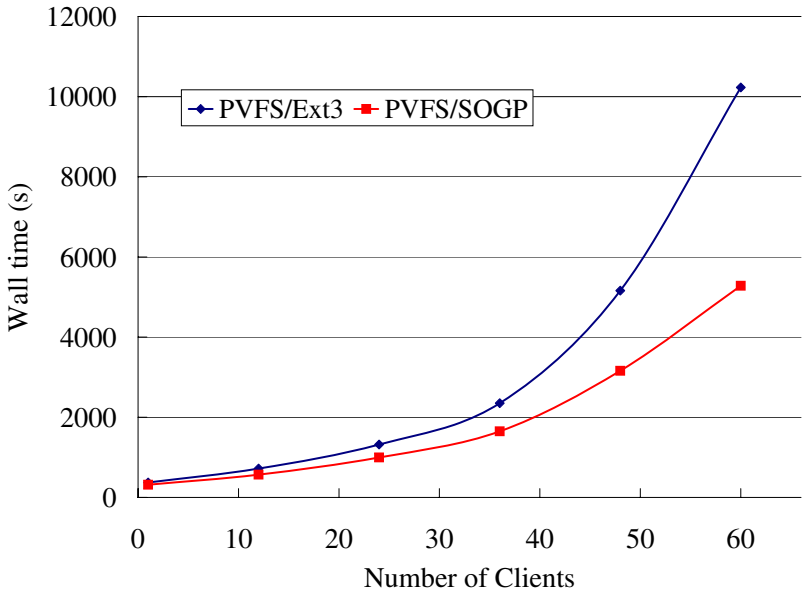


Figure 4.14: CP2K wall time with scaled problem size

PVFS/Ext3. This is expected because more "cubic" means more fragmented data accesses, while SOGP is designed to help improve the performance of fragmented data accesses.

For the scaled problem size case, the performance of CP2K in terms of wall time is shown in Figure 4.14. We observe approximately quadratic curves for both PVFS/SOGP and PVFS/Ext3 cases. The performance gain of PVFS/SOGP over PVFS/Ext3 ranges from 17% in the single client case to 94% in the 60 client case.

4.5.9 SIESTA performance comparison

The benchmark examines the time taken for one Self Consistent Field (SCF) iteration of Siesta. This includes a matrix diagonalization and some numerical real space grid integra-

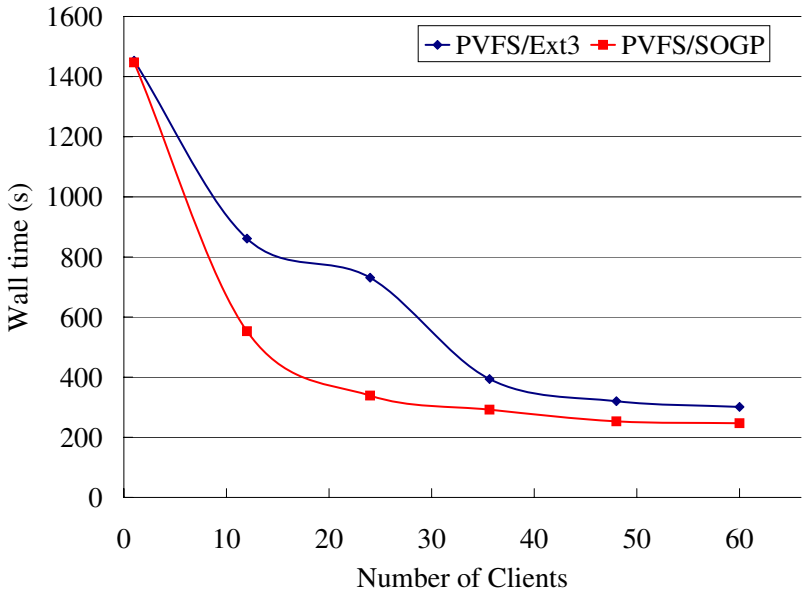


Figure 4.15: SIESTA benchmark wall time performance

tion. There are several modes of parallelism in SIESTA but the one tested was the most communications intensive. With one client, the PVFS/SOGP based system was marginally faster than PVFS/Ext3 for the Siesta benchmark (see Figure 4.15). However, with 12 clients PVFS/SOGP was 1.56 times faster than PVFS/Ext3 and 2.16 times faster with 24 clients. In this instance benchmarks were performed up to 60 processors. An interesting observation is that, with 24 clients, the performance of SIESTA based on PVFS/Ext3 displays some forms of unusual change that disturbs the smoothness of the corresponding curve, while this change is somehow "ignored" by the test based on PVFS/SOGP. A possible reason is that our test cases somehow generate more non-contiguous or partially overlapped I/O than normal with 24 clients, which results in adverse effect for Ext3, while those non-contiguous

or partially overlapped I/O access patterns are captured by SOGP and grouped into large segment I/O, smoothing out the potential performance disturbance.

4.6 Summary

In this chapter, we present the design and implementation of the first version of SOGP for a state-of-the-art parallel file system — PVFS. SOGP employs a segment I/O technique and a grouping based prefetching methodology to resolve some of the limitations existing in today's parallel file systems in how they interact with local storage and the impact of this method of interaction in the face of overlapping noncontiguous access, such as seen in many computational science post-processing and visualization applications. Using several parallel I/O benchmarks in different Linux-based cluster testbeds, we conclude that an SOGP-enhanced PVFS prototype system can significantly outperform a Linux-Ext3-based PVFS by up to 230% in terms of I/O bandwidth.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this dissertation, we have address several issues in metadata and data management in high performance file and storage systems. This chapter concludes the dissertation by summarizing our contribution first and then describing future research directions.

5.1 Contributions

The disseration addresses challenging problems for metadata and data management in high performance file and storage systems. To improve the metadata access performance, this work introduces a novel group-based prefetching algorithm (Nexus) specifically designed for metadata accesses in a decoupled metadata and data server cluster environment. In addition, to improve the data access performance in a representative parallel file system (PVFS) for multiple modern data intensive applications, we design and implemented a segment-structured on-disk grouping and prefetching (SOGP) algorithm in PVFS by taking advantage of a new access pattern that was never studied before.

5.1.1 Nexus: A novel metadata prefetching algorithm

After a detailed study and analysis of the difference on the size and access pattern of file data and file metadata, and its implications on the metadata access performance, this dissertation advocate the use of a novel weighted-graph based prefetching algorithm specifically designed for improving metadata access performance in a decoupled cluster storage environment. Aiming at the emerging MDS-cluster-based storage system architecture and exploiting the characteristic of metadata access, our prefetching algorithm distinguishes itself in several aspects. Specifically, we make the following contributions:

1. We collected the data usage information on a worlds' No. 9 supercomputer and performed a detailed study on the difference between file system metadata and regular file data in terms of size distribution. This study covers more than 8 million files and directory from the data center that serves more than one thousand of users from various research institute, national laboratories and industrial corporations.

2. Our new metadata prefetching algorithm is the first algorithm to take indirect successor relationship into prefetching considerations because of our unique comprehension that farther sight than the immediate successors has the potential to make wiser prefetching predictions. Comprehensive sensitivity study shows that the best metadata access performance can be achieved by looking at 5 immediate and subsequent successors rather than the conventional algorithms only looking at the immediate one.

3. Based on our observation on metadata size distribution on Franklin supercomputer, an aggressive prefetching approach is advocated for metadata prefetching algorithms. We justify the overhead of aggressive prefetching resulted from inaccurate prefetching and cache pollutions, and overcome the difficulty of high computational complexity for traditional single prefetching algorithms if adopted directly as multi-prefetching algorithms.

5.1.2 SOGP: Segment-structured On-disk Grouping and Prefetching Algorithm

With a distinct impression that the best local file system may not serve best as a data reposative for parallel file system, we identify and explore the information gap between these two layers in the distributed storage system hierarchy. We experiment our ideas on PVFS file system with several popular parallel I/O benchmarks and real work scientific applications, and find that an SOGP-enhanced PVFS prototype system can significantly outperform a Linux-Ext3-based PVFS. We enhance the data access performance through several improvement on PVFS based on the following findings.

1. We identify partially overlapped access pattern as our focus of study. Although similar noncontiguous patterns were reported in high performance computing community one decade ago, parallel file system and local file system architectures have not been focused on addressing these patterns.

2. We notice that, if we could recognize the on-disk data on the fly such that the future data accesses become sequential accesses, or keep a copy of the data in this more optimal organization, we could significantly improve the observed storage system bandwidth, and we could do so without changes to the rest of the storage system layers.

To exploit the potential performance gain based on these findings, we design and implemented the following techniques in SOGP.

1. An I/O technique called segment I/O to facilitate large, sequential disk I/O operations even when non-contiguous accesses are present. By grouping and coalescing small files or portions of a large file that are likely to be accessed together, we effectively reduce the number of noncontiguous I/O that would otherwise overwhelm the local file system.

2. We develop a locality based grouping and prefetching methodology, based on an existing file prefetching algorithm designed for local file system and a well-known stringent graph partitioning algorithm. This methodology can be used to detect partially overlapped non-contiguous accesses, such as seen in many computational science post-processing and visualization applications.

One high-performance local I/O software package in SOGP work for Parallel Virtual File System in the number of about 2000 C lines was released to Argonne National Laboratory in 2007 for potential integration into the production mode.

5.2 Future Work

During the investigation of I/O performance in parallel and distributed storage systems, we found several interesting research issues that have not been solved. This section briefly summarizes some of the open issues that need further investigations.

A relatively new class of distributed file system emerges in the past decade as object based file and storage systems. Examples includes Lustre file system, Ceph, and Panasas file system. They employ new storage architecture paradigms and new data/metadata organization techniques, even the definition of data and metadata could be totally different from what we have discussed so far. In these innovative distributed file systems designed to meet petabyte scale computing and storage system needs, the metadata and data I/O characteristic is worth revisiting, and hence the applicability of our Nexus algorithm for metadata prefetching on these new object based distributed file systems is worth exploring.

Our SOGP work also identifies possible enhancements for object storage systems as well, because the issue of lack of communication is also true for object based file and storage systems. More generally, this work points out the advantages of storing application data in more than one organization when ready-heavy workloads are anticipated. A worthwhile research plan could be to investigate how storing multiple representations of application data may be best managed, using SOGP and our grouping technology as a starting point.

LIST OF REFERENCES

[AB86] J. Archibald and J. L. Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model." *ACM Transactions on Computer Systems*, **4**(4), November 1986.

[AB02] D.D.E. Long A. Amer and R.C. Burns. "Group-based management of distributed file caches." *Proc. 22nd International Conference on Distributed Computing Systems*, p. 525, 2002.

[ACR96] Meenakshi Arunachalam, Alok N. Choudhary, and Brad Rullman. "Implementation and Evaluation of Prefetching in the Intel Paragon Parallel File System." In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pp. 554–559, Washington, DC, USA, 1996. IEEE Computer Society.

[AL01] Ahmed Amer and Darrell D. E. Long. "Noah: Low-cost file access prediction through pairs." *Proc. 20th IEEE Int'l Performance, Computing and Communications Conf.*, pp. 27–33, April 2001.

[ALB02] Ahmed Amer, Darrell D. E. Long, and Randal C. Burns. "Group-Based Management of Distributed File Caches." In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, p. 525, Washington, DC, USA, 2002. IEEE Computer Society.

[AMC07] Hiroshi Akiba, Kwan-Liu Ma, Jacqueline H. Chen, and Evatt R. Hawkes. "Visualizing Multivariate Volume Data from Turbulent Combustion Simulations." *Computing in Science and Engg.*, **9**(2):76–83, 2007.

[AW67] W. Anacker and Chu Ping Wang. "Performance Evaluation of Computing Systems with Memory Hierarchies." *Electronic Computers, IEEE Transactions on*, **EC-16**(6):764–773, Dec. 1967.

[BC05] D. (Daniele) Bovet and Marco Cesati. "Understanding the Linux kernel." *O'Reilly Associates, Inc.*, p. 923, 2005.

[BG03] John S. Bucy and G. R. Ganger. "The DiskSim Simulation Environment Version 3.0 Reference Manual." Technical report, Carnegie Mellon Univ., School of Computer Science, January 2003.

[Bry07] Randal E. Bryant. "Data-Intensive Supercomputing: The case for DISC." Technical Report CMU-CS-07-128, Carnegie Mellon University, May 2007.

[CFH94] Peter Corbett, Dror Feitselson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traverstat, and Parkson Wong. "MPI-IO: A parallel file I/O interface for MPI." Technical Report 19841 (87784), IBM T.J. Watson Research Center, Novemeber 1994.

[CFK96] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling." *ACM Trans. Comput. Syst.*, **14**(4):311–343, 1996.

[CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. "Practical Prefetching via Data Compression." *Proc. ACM Int'l Conf. on Management of Data*, pp. 257–266, May 1993.

[CLR00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. "PVFS: A Parallel File System for Linux Clusters." In *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 317–327, Atlanta, GA, 2000. USENIX Association.

[CLR01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter Section 22.5, pp. 552–557. MIT Press and McGraw-Hill, second edition edition, 2001.

[DJC07a] Xiaoning Ding, Song Jiang, and Feng Chen. "A buffer cache management scheme exploiting both temporal and spatial localities." *Trans. Storage*, **3**(2):5, 2007.

[DJC07b] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhan. "DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch." In *USENIX Annual Technical Conference*, pp. 261–274. USENIX, 2007.

[DW07] Ananth Devulapalli and Pete Wyckoff. "File Creation Strategies in a Distributed Metadata File System." *Proc. 21st IEEE Int'l Parallel and Distributed Processing Symp.*, pp. 1–10, 2007.

[DWA94] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. "Cooperative Caching: Using Remote Client Memory to Improve File System Performance." Technical report, Univ. of California, Berkeley, December 1994.

[GA94] J. Griffionen and R. Appleton. "Reducing File System Latency using a Predictive Approach." *Proc. USENIX Summer 1994 Technical Conf.*, June 6–10 1994.

[GA03] Minaxi Gupta and Mostafa Ammar. "A Novel Multicast Scheduling Scheme for Multimedia Servers with Variable Access Patterns." *Proc. IEEE Int'l Conf. on Communications*, May 2003.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system." *Proc. 9th ACM Symp. on Operating systems principles*, pp. 29–43, 2003.

[GZJ06] Peng Gu, Yifeng Zhu, Hong Jiang, and Jun Wang. "Nexus: A Novel Weighted-Graph-Based Prefetching Algorithm for Metadata Servers in Petabyte-Scale Storage Systems." *Proc. 6th IEEE Int'l Symp. on Cluster Computing and the Grid*, pp. 409–416, 2006.

[Had00] Ibrahim F. Haddad. "PVFS: A Parallel Virtual File System for Linux Clusters." *Linux J.*, p. 5, 2000.

[HO95] John H. Hartman and John K. Ousterhout. "The Zebra striped network file system." *ACM Trans. Comput. Syst.*, **13**(3):274–310, 1995.

[IEE05] "IEEE 802.3 standard." 2005.

[IOR] "IOR Benchmark." http://sourceforge.net/projects/ior-sio/.

[JCE95] Jr. James V. Huber, Andrew A. Chien, Christopher L. Elford, David S. Blumenthal, and Daniel A. Reed. "PPFS: a high performance portable parallel file system." In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pp. 385–394, New York, NY, USA, 1995. ACM.

[KE91] David Kotz and Carla Schlatter Ellis. "Practical prefetching techniques for parallel file systems." In *PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems*, pp. 182–189, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[KE93] David Kotz and Carla Schlatter Ellis. "Practical Prefetching Techniques for Multiprocessor File Systems." *J. Distributed and Parallel Databases*, **1**(1):33–51, January 1993.

[KL91] Alexander C. Klaiber and Henry M. Levy. "An Architecture for Software-Controlled Data Prefetching." *Proc. ACM 18th Annual Int. Symp. on Computer Architecture*, **19**(3):43–53, 1991.

[KL99] Thomas M. Kroeger and Darrell D. E. Long. "The Case for Efficient File Access Pattern Modeling." In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, p. 14, Washington, DC, USA, 1999. IEEE Computer Society.

[KL01] Tom M. Kroeger and Darrell D. E. Long. "Design and Implementation of a Predictive File Prefetching Algorithm." In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pp. 105–118, Berkeley, CA, USA, 2001. USENIX Association.

[Knu85] Donald E. Knuth. "An Analysis of Optimal Caching." *J. Algorithms*, **6**:181–199, 1985.

[Kot97] David Kotz. "Disk-directed I/O for MIMD multiprocessors." *ACM Trans. Comput. Syst.*, **15**(1):41–74, 1997.

[KSK08] Mahmut Kandemir, Seung Woo Son, and Mustafa Karakoy. "Improving I/O performance of applications through compiler-directed code restructuring." In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pp. 1–16, Berkeley, CA, USA, 2008. USENIX Association.

[LD97] Hui Lei and Dan Duchamp. "An Analytical Approach to File Prefetching." *Proc. USENIX Annual Technical Conf.*, January 1997.

[Lin07] "The Linux Kernel Archives." 2007.

[LM99] Chi-Keung Luk and Todd C. Mowry. "Automatic Compiler-Inserted Prefetching for Pointer-Based Applications." *IEEE Trans. Comput.*, **48**(2):134–141, 1999.

[LSK01] Yoon-Young Lee, Dae-Wha Seo, and Chei-Yol Kim. "Table-Comparison Prefetching in VIA-based Parallel File System." In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, p. 163, Washington, DC, USA, 2001. IEEE Computer Society.

[Lus04] "Lustre." http://www.lustre.org, June 2004.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. "A low-bandwidth network file system." *Proc. 8th ACM Symp. on Operating systems principles*, pp. 174–187, 2001.

[MDK96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. "Automatic compiler-inserted I/O prefetching for out-of-core applications." In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pp. 3–17. USENIX Association, 1996.

[Mic08] Sun Microsystems. "Lustre FAQ." http://wiki.lustre.org/index.php?title=Lustre_FAQ#Why_did_Lustre_choose_ext3.3F_Do_you_ever_plan_to_support_others.3F, April 2008.

[MKL06] Gokhan Memik, Mahmut T. Kandemir, Wei-Keng Liao, and Alok Choudhary. "Multicollective I/O: A technique for exploiting inter-file access patterns." *Trans. Storage*, **2**(3):349–369, 2006.

[MR02] Tara M. Madhyastha and Daniel A. Reed. "Learning to Classify Parallel Input/Output Access Patterns." *IEEE Transactions on Parallel and Distributed Systems*, **13**(8):802–813, August 2002.

[MS96] Steven A. Moyer and V. S. Sunderam. "Characterizing concurrency control performance for the PIOUS parallel file system." *J. Parallel Distrib. Comput.*, **38**(1):81–91, 1996.

[NKP96] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. "File-Access Characteristics of Parallel Scientific Workloads." *IEEE Transactions on Parallel and Distributed Systems*, **7**(10):1075–1089, October 1996.

[NSM04] David Nagle, Denis Serenyi, and Abbie Matthews. "The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage." In *SC'2004 Conference CD*, Pittsburgh, PA, November 2004. IEEE/ACM SIGARCH.

[ORR04] Ekow Otoo, Doron Rotem, and Alexandru Romosan. "Optimal File-Bundle Caching Algorithms for Data-Grids." *Proc. ACM/IEEE Conf. on Supercomputing*, p. 6, 2004.

[PAL03] Jehan-François Pâris, Ahmed Amer, and Darrell D. E. Long. "A stochastic approach to file access prediction." *Proc. Int'l workshop on Storage network architecture and parallel I/Os*, pp. 36–40, 2003.

[PG94] R. Hugo Patterson and Garth A. Gibson. "Exposing I/O concurrency with informed prefetching." In *PDIS '94: Proceedings of the third international conference on on Parallel and distributed information systems*, pp. 7–16, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PGS93] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. "A status report on research in transparent informed prefetching." *ACM Operating Systems Review*, **27**(2):21–34, 1993.

[pio] "Parallel I/O Consortium." http://www-unix.mcs.anl.gov/∼rross/pio-benchmark/.

[RFL06] Rob Ross, Evan Felix, Bill Loewe, Lee Ward, James Nunez, John Bent, Ellen Salmon, and Gary Grider. "File Systems and I/O Research Workshop HECIWG FSIO 2006 Report." Technical report, HECRTF, HECIWG, 2006.

[RKS02] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. "A Framework for Evaluating Storage System Security." In *FAST*, pp. 15–30, 2002.

[RLA00] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. "A Comparison of File System Workloads." *Proc. USENIX Annual Technical Conf.*, pp. 41–54, June 18–23 2000.

[SAG02] José M Soler, Emilio Artacho, Julian D Gale, Alberto García, Javier Junquera, Pablo Ordejón, and Daniel Sánchez-Portal. "The SIESTA method for ab initio order-N materials simulation." *Journal of Physics: Condensed Matter*, **14**(11):2745–2779, 2002.

[Sch03] P. Schwan. "Lustre: Building a file system for 1000-node clusters." *Proc. 2003 Linux Symp.*, 2003.

[SD00] Jonas Skeppstedt and Michel Dubois. "Compiler Controlled Prefetching for Multiprocessors Using Low-Overhead Traps and Prefetch Engines." *J. Parallel Distrib. Comput*, **60**(5):585–615, 2000.

[SH02] F. Schmuck and R. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." pp. 231–244, Jan., 2002.

[Sho03] Frank Shorter. *"Design and Analysis of A Performance Evaluation Standard for Parallel File Systems."*. Master's thesis, Clemson University, 2003.

[SSS99] Elizabeth Shriver, Christopher Small, and Keith A. Smith. "Why does file system prefetching work?" In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pp. 6–6, Berkeley, CA, USA, 1999. USENIX Association.

[Tom97] Andrew Tomkins. "Practical and theoretical in prefetching and caching." *PhD dissertation*, 1997.

[WBM06a] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. "Ceph: a scalable, high-performance distributed file system." In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[WBM06b] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. "CRUSH: controlled, scalable, decentralized placement of replicated data." *Proc. 2006 ACM/IEEE Conf. on Supercomputing*, p. 122, 2006.

[WK03] Yijian Wang and David Kaeli. "Profile-guided I/O partitioning." In *Proceedings of the 2003 International Conference on Supercomputing (ICS-03)*, pp. 252–260, New York, June 23–26 2003. ACM Press.

[WPB04] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. "Dynamic Metadata Management for Petabyte-Scale File Systems." *Proc. ACM/IEEE Conf. on Supercomputing*, pp. 4–4, nov 2004.

[WS06] Chaoli Wang and Han-Wei Shen. "LOD Map - A Visual Interface for Navigating Multiresolution Volume Visualization." *IEEE Transactions on Visualization and Computer Graphics*, **12**(5):1029–1036, 2006.

[WUA08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. "Scalable Performance of the Panasas Parallel File System." pp. 17–33, Feb 2008.

[WWP03] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar Pandac. "PVFS over InfiniBand: Design and Performance Evaluation." In *Proceedings of the 2003 International Conference on Parallel Processing (32th ICPP'03)*, pp. 125–132, Kaohsiung, Taiwan, October 2003. IEEE Computer Society.

[WXH04] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. Miller, D. Long, and T. McLarty. "File System Workload Analysis for Large Scale Scientific Computing Applications." Jan 2004.

[YLP05] Weikuan Yu, Shuang Liang, and Dhabaleswar K. Panda. "High performance support of parallel virtual file system (PVFS2) over Quadrics." In Arvind and Larry Rudolph, editors, *ICS*, pp. 323–331. ACM, 2005.

[YS93] H.C. Young and E.J. Shekita. "An intelligent I-cache prefetch mechanism." *Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on*, pp. 44–49, Oct 1993.

[ZJW04] Yifeng Zhu, Hong Jiang, and Jun Wang. "Hierarchical Bloom filter Arrays (HBA):a novel, scalable metadata management system for large cluster-based storage." *Proc. IEEE Int'l Conf. on Cluster Computing*, pp. 165–174, Oct 2004.

[ZL07] Xiaotong Zhuang and Hsien-Hsin S. Lee. "Reducing Cache Pollution via Dynamic Data Prefetch Filtering." *IEEE Trans. Computers*, **56**:18–31, Jan 2007.